Madrid
Febrero 2001

Aprenda Matlab 5.3

como si estuviera en primero

Javier García de Jalón · José Ignacio Rodríguez · Alfonso Brazález



Escuela Técnica Superior
de Ingenieros Industriales

Universidad Politécnica de Madrid

ÍNDICE

1. INTRODUCCIÓN

1.1. Acerca de este Manual

Las primeras versiones de este manual estuvieron dirigidas a los alumnos de ***Informática 1*** en la Escuela Superior de Ingenieros Industriales de San Sebastián. Esta asignatura se cursa en el primer semestre de la carrera y el aprendizaje de MATLAB constituía la primera parte de la asignatura. Se trataba pues de un manual introductorio de una aplicación que, para muchos alumnos, iba a constituir su primer contacto con los ordenadores y/o con la programación.

Desde el curso 2000-2001, este manual se ha adaptado a las asignaturas de ***Matemáticas de la Especialidad (Mecánica-Máquinas)*** (Plan 1976) y ***Álgebra II*** (Plan 2000) en la Escuela Técnica Superior de Ingenieros Industriales de la Universidad Politécnica de Madrid.

Por encima de las asignaturas citadas, este manual puede ser útil a un público más amplio, que incluye a alumnos de cursos superiores de las Escuelas de Ingeniería Industrial, a alumnos de Tercer Ciclo y a profesores que quieren conocer más de cerca las posibilidades que tendría MATLAB en sus asignaturas.

Se ha pretendido llegar a un equilibrio entre el detalle de las explicaciones, la amplitud de temas tratados y el número de páginas. En algunos casos, junto con las instrucciones introducidas por el usuario se incluye la salida de MATLAB; en otros casos no se incluye dicha salida, pero se espera que el lector disponga de un PC con MATLAB y vaya introduciendo esas instrucciones a la vez que avanza en estas páginas. En muchas ocasiones se anima al lector interesado a ampliar el tema con la ayuda del programa (toda la documentación de MATLAB está disponible *on-line* a través del ***Help***). En cualquier caso recuérdese que la informática moderna, más que en "saber" consiste en "saber encontrar lo que se necesita" en pocos segundos.

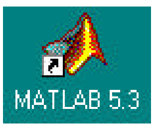
1.2. El programa MATLAB

MATLAB es el nombre abreviado de "MATrix LABoratory". MATLAB es un programa para realizar cálculos numéricos con ***vectores*** y ***matrices***. Como caso particular puede también trabajar con números escalares, tanto reales como complejos. Una de las capacidades más atractivas es la de realizar una amplia variedad de ***gráficos*** en dos y tres dimensiones. MATLAB tiene también un lenguaje de programación propio. Este manual hace referencia a la versión 5.3 de este programa, aparecida a comienzos de 1999.

MATLAB es un gran programa de para cálculo técnico y científico. Para ciertas operaciones es muy rápido, cuando puede ejecutar sus funciones en código nativo con los tamaños más adecuados para aprovechar sus capacidades de vectorización. En otras aplicaciones resulta bastante más lento que el código equivalente desarrollado en C/C++ o Fortran. Sin embargo, siempre es una magnífica herramienta de alto nivel para desarrollar aplicaciones técnicas, fácil de utilizar y que aumenta la productividad de los programadores respecto a otros entornos de desarrollo.

MATLAB dispone de un código básico y de varias librerías especializadas (***toolboxes***). En estos apuntes se hará referencia exclusiva al código básico.

MATLAB se puede arrancar como cualquier otra aplicación de ***Windows 95/98/NT***, clicando dos veces en el icono correspondiente en el escritorio o por medio del menú ***Start***). Al arrancar MATLAB se abre una ventana del tipo de la indicada en la Figura 1.



En la ventana inicial se sugieren ya algunos comandos para el usuario inexperto que quiere echar un vistazo a la aplicación. En dicha ventana aparece también el ***prompt*** (aviso) característico de MATLAB (»). Esto quiere decir que el programa está preparado para recibir instrucciones.

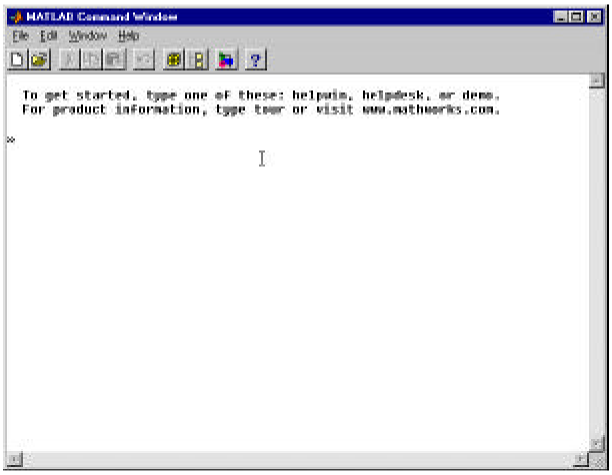


Figura 1. Ventana inicial de MATLAB 5.3.

Puede hacerse que aparezca un saludo inicial personalizado por medio de un ***fichero de comandos*** personal que se ejecuta de modo automático cada vez que se entra en el programa (el fichero ***startup.m***, que debe estar en un directorio determinado, por ejemplo ***C:\Matlab***. Ver Apartado 1.4.2, en la página 7).

Para apreciar desde el principio la potencia de MATLAB, se puede comenzar por escribir la siguiente línea, a continuación del *prompt*. Al final hay que pulsar ***intro***.

```
» A=rand(6), B=inv(A), B*A
A =
    0.9501    0.4565    0.9218    0.4103    0.1389    0.0153
    0.2311    0.0185    0.7382    0.8936    0.2028    0.7468
    0.6068    0.8214    0.1763    0.0579    0.1987    0.4451
    0.4860    0.4447    0.4057    0.3529    0.6038    0.9318
    0.8913    0.6154    0.9355    0.8132    0.2722    0.4660
    0.7621    0.7919    0.9169    0.0099    0.1988    0.4186
B =
    5.7430    2.7510    3.6505    0.1513   -6.2170   -2.4143
   -4.4170   -2.5266   -1.4681   -0.5742    5.3399    1.5631
   -1.3917   -0.6076   -2.1058   -0.0857    1.5345    1.8561
   -1.6896   -0.7576   -0.6076   -0.3681    3.1251   -0.6001
   -3.6417   -4.6087   -4.7057    2.5299    6.1284    0.9044
    2.7183    3.3088    2.9929   -0.1943   -5.1286   -0.6537
ans =
    1.0000    0.0000         0    0.0000    0.0000   -0.0000
    0.0000    1.0000    0.0000    0.0000   -0.0000    0.0000
         0         0    1.0000   -0.0000   -0.0000    0.0000
    0.0000         0   -0.0000    1.0000   -0.0000    0.0000
   -0.0000    0.0000   -0.0000   -0.0000    1.0000    0.0000
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
```

En realidad, en la línea de comandos anterior se han escrito tres instrucciones diferentes, separadas por comas. Como consecuencia, la respuesta del programa tiene tres partes también, cada una de ellas correspondiente a una de las instrucciones. Con la primera instrucción se define una matriz cuadrada (6x6) llamada **A**, cuyos elementos son números aleatorios entre cero y uno (aunque aparezcan sólo 4 cifras, han sido calculados con 16 cifras). En la segunda instrucción se define una matriz **B** que es igual a la inversa de **A**. Finalmente se ha multiplicado **B** por **A**, y se comprueba que el resultado es la matriz unidad[1].

Con grandes matrices o grandes sistemas de ecuaciones MATLAB obtiene toda la potencia del ordenador. Por ejemplo, las siguientes instrucciones permiten calcular la potencia de cálculo del ordenador en Megaflops (millones de operaciones aritméticas por segundo). En la primera línea se

[1] Al invertir la matriz y al hacer el producto posterior se han introducido pequeños errores numéricos de redondeo en el resultado, lo cual hace que no todos los elementos cero del resultado aparezcan de la misma forma.

crean tres matrices de tamaño 500×500, las dos primeras con valores aleatorios y la tercera con valores cero. La segunda línea pone a cero el contador de operaciones aritméticas, toma tiempos, realiza el producto de matrices, vuelve a tomar tiempos y calcula el número de millones de operaciones realizadas. La tercera línea calcula los Megaflops por segundo, para lo cual utiliza la función ***etime()*** que calcula el tiempo transcurrido entre dos instantes definidos por dos llamadas a la función ***clock***.

```
» A=rand(500); B=rand(500); C=zeros(500);
» flops(0); tini=clock; C=B*A; tend=clock; mflops=flops/1000000;
» mflops/etime(tend,tini)
```

Otro de los puntos fuertes de MATLAB son los gráficos, que se verán con más detalle en una sección posterior. A título de ejemplo, se puede teclear la siguiente línea y pulsar ***intro***:

```
» x=-4:.01:4; y=sin(x); plot(x,y), grid, title('Función seno(x)')
```

En la Figura 2 se puede observar que se abre una nueva ventana en la que aparece representada la función ***sin(x)***. Esta figura tiene un título "Función seno(x)" y una cuadrícula o "grid". En realidad la línea anterior contiene también varias instrucciones separadas por comas o puntos y comas. En la primera se crea un vector **x** con 801 valores reales entre -4 y 4, separados por una centésima. A continuación se crea un vector **y**, cada uno de cuyos elementos es el seno del correspondiente elemento del vector **x**. Después se dibujan los valores de **y** en ordenadas frente a los de **x** en abscisas. Las dos últimas instrucciones establecen la cuadrícula y el título.

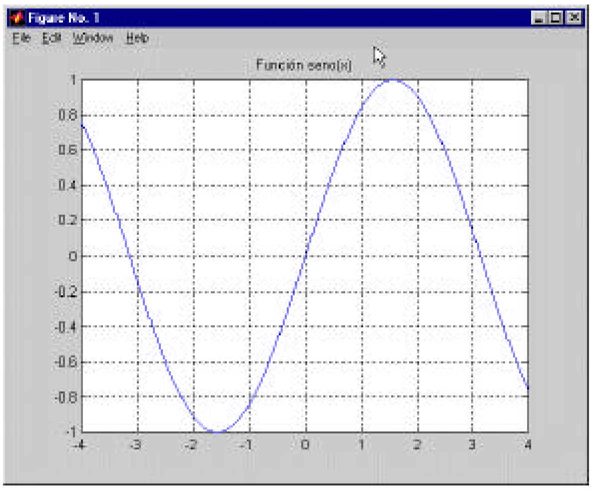


Figura 2. Gráfico de la función ***seno(x)***.

Un pequeño aviso antes de seguir adelante. Es posible recuperar comandos anteriores de MATLAB y moverse por dichos comandos con el ratón y con las teclas-flechas ↑ y ↓. Al pulsar la primera de dichas flechas aparecerá el comando que se había introducido inmediatamente antes. De modo análogo es posible moverse sobre la línea de comandos con las teclas ← y →, ir al principio de la línea con la tecla ***Inicio***, al final de la línea con ***Fin***, y borrar toda la línea con ***Esc***. Recuérdese que sólo hay una línea activa (la última).

Para borrar todas las salidas anteriores de MATLAB y dejar limpia la ventana principal se pueden utilizar las funciones ***clc*** y ***home***. La función ***clc*** (*clear console*) elimina todas las salidas anteriores, mientras que ***home*** las mantiene, pero lleva el ***prompt*** (**»** **)** a la primera línea de la ventana.

Si se desea salir del programa, basta teclear los comandos ***quit*** o ***exit***, o bien elegir ***Exit*** MATLAB en el menú ***File*** (también se puede utilizar el ***Alt+F4*** de todas las aplicaciones de ***Windows***).

1.3. Uso del *Help*

MATLAB 5.3 dispone de un excelente ***Help*** con el que se puede encontrar la información que se desee. La Figura 3 muestra las distintas opciones que aparecen en el menú ***Help***.

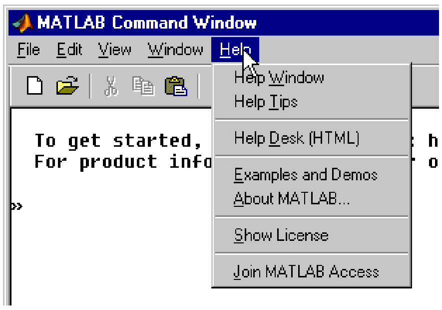


Figura 3. Menú ***Help*** de Matlab.

1. ***Help Window***, Se abre la ventana de la Figura 4, en la que se puede buscar ayuda sobre la función o el concepto que se desee.
2. ***Help Tips.*** Ofrece ideas prácticas para utilizar la ayuda.
3. ***Help Desk***. Se abre un browser de Internet (***Netscape Communicator***, en el caso de la Figura 5) que permite acceder a toda la información sobre MATLAB en formato HTML. Esta información es equivalente a los manuales impresos del programa. Desde la parte inferior de esta página, mediante el enlace ***Online Manuals (in PDF)*** se puede acceder a la versión ****.pdf*** (*Portable Document Format*) de los manuales de MATLAB. Este formato es menos adecuado para consultar sobre la pantalla que el HTML, pero mucho más adecuado para imprimir y revisar luego sobre papel. El formato ****.pdf*** requiere del programa gratuito ***Adobe Acrobat Reader 3.0*** o una versión superior.
4. ***Examples and Demos***. Se abre una ventana que da acceso a un buen número de ejemplos resueltos con MATLAB, cuyos resultados se presentan gráficamente de diversas formas. Es bastante interesante recorrer estos ejemplos para hacerse idea de las posibilidades del programa. Es asimismo muy instructivo.

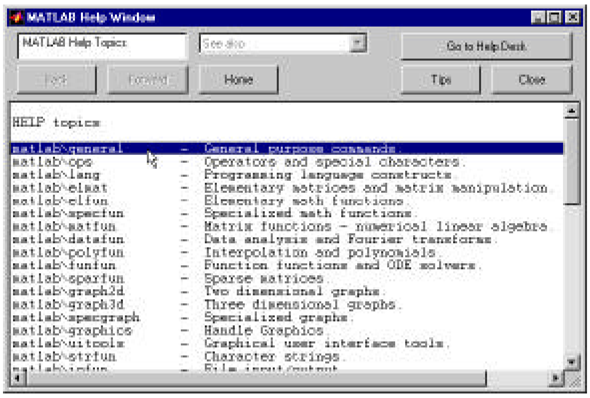


Figura 4. Ventana inicial de ***Help Window***.



Figura 5. Ventana inicial de ***Help Desk***.

Además, se puede también recurrir al ***Help*** desde la línea de comandos. Se aconseja hacer prácticas al respecto. Por ejemplo, obsérvese la respuesta a los siguientes usos del comando ***help***:

```
» help
» help lang
```

El comando ***helpwin*** seguido de un nombre de comando muestra la información correspondiente a ese comando en la ventana ***Help Window*** (ver Figura 4), incluyendo también comandos similares sobre los que se ofrece ayuda (lista desplegable ***See Also***).

El comando ***doc*** seguido de un nombre de comando muestra la información correspondiente a ese comando a través de ***Netscape Navigator*** o ***Internet Explorer***, en formato HTML.

1.4. El entorno de trabajo de MATLAB

El entorno de trabajo de MATLAB ha mejorado mucho a partir de la versión 5.0, haciéndose mucho más gráfico e intuitivo. Los componentes más importantes del entorno de trabajo de MATLAB son el *editor de caminos de búsqueda* (***Path Browser***), el *editor y depurador de errores* (***Editor & Debugger***) y el *visualizador del espacio de trabajo* (***Workspace Browser***). A continuación se describen brevemente estos componentes.

Utilizar MATLAB y desarrollar programas para MATLAB es mucho más fácil si se conoce bien este entorno de trabajo. Es por ello muy importante leer con atención las secciones que siguen.

1.4.1. *PATH BROWSER*: ESTABLECER EL CAMINO DE BÚSQUEDA (*SEARCH PATH*)

MATLAB puede llamar a una gran variedad de funciones, tanto del propio programa como programadas por los usuarios. A veces puede incluso haber funciones distintas que tienen el mismo nombre. Interesa saber cuáles son las reglas que determinan qué función o qué fichero ***.m***[2] es el que se va a ejecutar cuando su nombre aparezca en una línea de comandos del programa. Esto queda determinado por el *camino de búsqueda* (***search path***) que el programa utiliza cuando encuentra el nombre de una función.

El ***search path*** de MATLAB es una lista de directorios que se puede ver y modificar a partir de la línea de comandos, o utilizando el ***Path Browser***. El comando ***path*** hace que se escriba el ***search path*** de MATLAB (el resultado depende de en qué directorio esté instalado MATLAB):

```
» path

        MATLABPATH

    c:\matlab\toolbox\local
    c:\matlab\toolbox\matlab\datafun
    c:\matlab\toolbox\matlab\elfun
    .... (por brevedad se omiten muchas de las líneas de salida)
    c:\matlab\toolbox\matlab\dde
    c:\matlab\toolbox\matlab\demos
    c:\matlab\toolbox\wintools
```

Para ver cómo se utiliza el ***search path*** supóngase que se utiliza la palabra ***nombre1*** en un comando. El proceso que sigue el programa para tratar de conocer qué es ***nombre1*** es el siguiente:

1. Comprueba si ***nombre1*** es una variable previamente definida por el usuario.
2. Comprueba si ***nombre1*** es una función interna o intrínseca de MATLAB.
3. Comprueba si ***nombre1*** es una *sub-función* o una función *privada* del usuario (ver Apartado 5.3).
4. Comprueba si hay un fichero llamado ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en el ***directorio actual***, cuyo contenido se obtiene con el comando ***dir***. El ***directorio actual*** se cambia con el comando ***cd***.
5. Comprueba si hay ficheros llamados ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en los directorios incluidos en el ***search path*** de MATLAB.

---

[2] Los ficheros ***.m*** son ficheros ASCII que definen funciones o contienen comandos de MATLAB. A lo largo de estas páginas se volverá sobre estos ficheros con mucho detenimiento.

Estos pasos se realizan por el orden indicado. En cuanto se encuentra lo que se está buscando se detiene la búsqueda y se utiliza el fichero que se ha encontrado. Conviene saber que, a igualdad de nombre, los ficheros ***.mex*** tienen precedencia sobre los ficheros ***.m*** que están en el mismo directorio.

El concepto de ***directorio actual*** es importante en MATLAB. Para cambiar de *directorio actual* se utiliza el comando ***cd*** (de *change directory*), seguido del nombre del directorio, para el cual se puede utilizar un ***path*** absoluto (por ejemplo ***cd C:\Matlab\Ejemplos***) o relativo (***cd Ejemplos***). Para subir un nivel en la jerarquía de directorios se utiliza el comando ***cd ..***, y ***cd ../..*** para subir dos niveles. MATLAB permite utilizar tanto la barra normal (/) como la barra invertida (\), indistintamente.

El ***Path Browser*** es el programa que ayuda a definir la lista de directorios donde MATLAB debe buscar los ficheros de comandos y las funciones, tanto del sistema como de usuario. Con el comando ***Set Path*** del menú ***File*** aparece el cuadro de diálogo de la Figura 6, en el cual se muestra la lista de directorios en la que MATLAB buscará. Para añadir (o quitar) un directorio a esta lista se debe ejecutar el comando ***Add to Path*** (o ***Remove Path***) en el menú ***Path*** de dicho cuadro de diálogo, con lo cual aparece un nuevo cuadro de diálogo mostrado en el Figura 7. El nuevo directorio se puede añadir al comienzo o final de la lista. Como ya se ha dicho el orden de la lista es muy importante, porque refleja el orden de la búsqueda: si una función está en dos directorios, se utilizará la que primero se encuentre.

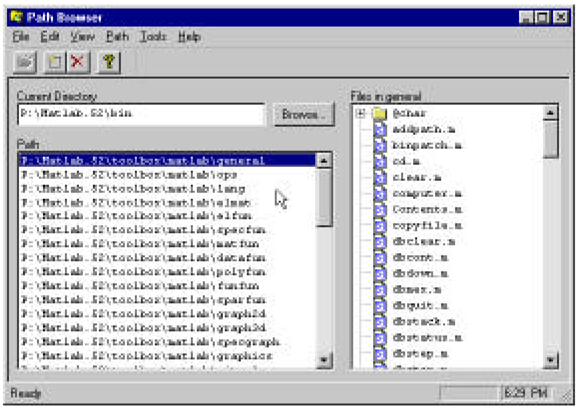


Figura 6. ***Path Browser***.

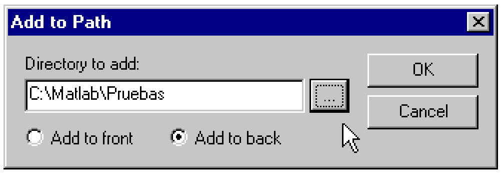


Figura 7. Añadir un directorio al ***Path***.

Para incluir desde la línea de comandos de MATLAB un directorio nuevo al comienzo del ***search path*** sin utilizar el ***Path Browser***, se puede utilizar también el comando ***path***, que concatena dos listas de directorios (sólo se deben utilizar directorios que realmente existan en el PC), como por ejemplo:

```
» path('c:\mat\matlab', path)
```
[3]

mientras que para añadir el nuevo directorio al final de la lista, se utiliza:

```
» path(path, 'c:\mat\practicas')
```

El comando ***addpath*** permite añadir uno o más directorios al ***search path***. Su forma general puede verse en los siguientes ejemplos:

```
» addpath 'c:\Matlab' 'c:\Temp' -end
» addpath 'c:\Matlab\Pruebas' 'c:\Temp\Pruebas' -begin
```

donde la opción por defecto (cuando no se pone ni *-begin* ni *-end*) es añadir al comienzo de la lista. Después de ejecutar estos comandos conviene comprobar cómo ha quedado modificado el ***search path*** (recuérdese que los directorios deben existir en realidad).

No es difícil borrar las líneas que se han introducido: por una parte, los cambios no son permanentes y dejarán de surtir efecto al salir de MATLAB y volver a entrar (salvo que se guarden

[3] El comando ***path*** dentro del paréntesis de la función devuelve la lista de directorios anterior.

como opciones estables). Además se puede utilizar el comando ***rmpath*** (de *remove path*), al que se le pasan la lista de directorios a eliminar del ***search path***. Por ejemplo, el comando:

```
» rmpath 'c:\Matlab' 'c:\Temp'
```

borra del ***search path*** los dos directorios indicados.

1.4.2. FICHEROS *MATLABRC.M*, *STARTUP.M* Y *FINISH.M*

El ***search path*** inicial o *por defecto* de MATLAB está contenido en un fichero llamado ***matlabrc.m***, en el sub-directorio ***toolbox\local***. Este fichero contiene también muchos otros parámetros de inicialización y es, por ejemplo, el responsable del mensaje que aparece al arrancar el programa. Este fichero se ejecuta automáticamente al arrancar MATLAB.

En las instalaciones de MATLAB en red, ***matlabrc.m*** es un fichero controlado por el administrador del sistema. Una de las cosas que hace este fichero es ver si en algún directorio del ***search path*** existe otro fichero llamado ***startup.m***, y en caso de que exista lo ejecuta. Esto abre la posibilidad de que cada usuario arranque MATLAB de una forma personalizada. En el ***search path*** de MATLAB contiene un directorio local tal como ***G:\Matlab***. Como éste es un directorio personal de cada usuario (por serlo la unidad ***G:***), si en dicho directorio se coloca un fichero llamado ***startup.m*** las instrucciones contenidas en dicho fichero se ejecutarán automáticamente cada vez que arranque MATLAB.

Un posible contenido de este fichero puede ser el siguiente (se sugiere crearlo con ***Notepad***):

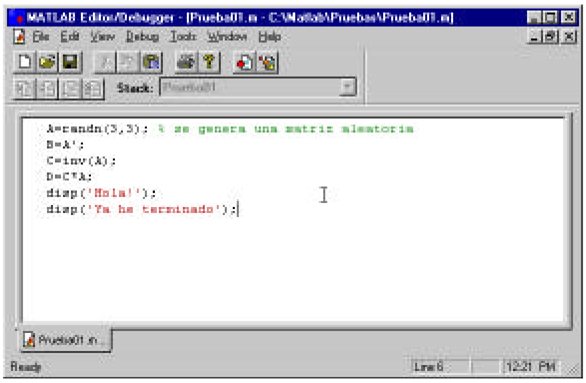
```
» format compact
» addpath 'g:\Matlab\Practicas' -end
» disp('¡Hola!')
```

Se puede crear fichero en el directorio indicado y probar a arrancar MATLAB. Si el saludo ***¡Hola!*** se sustituye por un saludo más personal (por ejemplo, incluyendo el nombre), se comprobará lo explicado previamente. Es muy aconsejable crear este fichero.

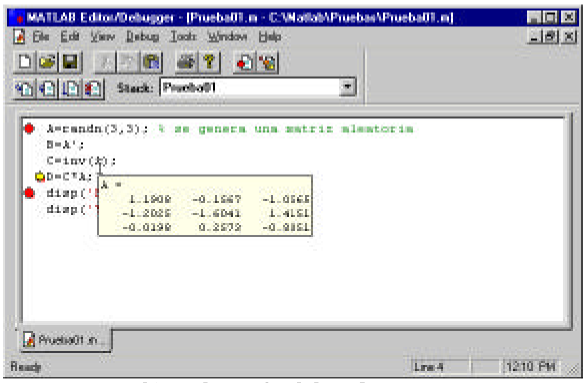
De forma análoga, al abandonar la ejecución de MATLAB con el comando ***quit*** se ejecuta automáticamente el fichero ***finish.m***, siempre que se encuentre en alguno de los directorios del ***search path***. Este fichero se puede utilizar por ejemplo para guardar el espacio de trabajo de MATLAB (ver Apartados 1.4.4 y 1.6, en las páginas 9 y 11, respectivamente) y poder continuar en otro momento a partir del punto en el que se abandonó el trabajo, por ejemplo cerrando el programa.

1.4.3. *EDITOR&DEBUGGER*: EDITOR DE FICHEROS Y DEPURADOR DE ERRORES

En MATLAB tienen particular importancia los ya citados ***ficheros-M*** (o ***M-files***). Son ficheros de texto ASCII, con la extensión ****.m***, que contienen ***conjuntos de comandos*** o ***definición de funciones*** (estos últimos son un poco más complicados y se verán más adelante). La importancia de estos *ficheros-M* es que al teclear su nombre en la línea de comandos y pulsar ***Intro***, se ejecutan uno tras otro todos los comandos contenidos en dicho fichero. El poder guardar instrucciones y grandes matrices en un fichero permite siempre ahorrar mucho trabajo de tecleado.



a) Creación de un fichero de comandos.



b) Utilización del ***Debugger***.

Figura 8. El ***Editor/Debugger*** de MATLAB.

Aunque los ficheros ****.m*** se pueden crear con cualquier editor de ficheros ASCII tal como ***Notepad***, MATLAB dispone de un ***editor*** que permite tanto crear y modificar estos ficheros, como ejecutarlos paso a paso para ver si contienen errores (proceso de ***Debug*** o depuración). La Figura 8a muestra la ventana principal del ***Editor/Debugger***, en la que se ha tecleado un *fichero-M* llamado ***Prueba01.m***, que contiene seis comandos[4]. El ***Editor*** muestra con diferentes colores los diferentes tipos o elementos constitutivos de los comandos (en *verde* los comentarios, en *rojo* las cadenas de caracteres, etc.). El ***Editor*** se preocupa también de que las comillas o paréntesis que se abren, no se queden sin el correspondiente elemento de cierre.

La Figura 8b corresponde a una ejecución de este fichero de comandos controlada con el ***Debugger***. Dicha ejecución se comienza eligiendo el comando ***Run*** en el menú ***Tools*** o tecleando el nombre del fichero en la línea de comandos. Los puntos rojos que aparecen en el margen izquierdo son ***breakpoints*** (puntos en los que se detiene la ejecución de programa); la flecha amarilla indica la sentencia en que está detenida la ejecución (antes de ejecutar dicha sentencia); cuando el cursor se coloca sobre una variable (en este caso sobre la matriz **A**) aparece una pequeña ventana con los valores numéricos de esa variable.

Puede apreciarse que en la Figura 8b está activada la segunda barra de herramientas, que corresponde al ***Debugger***. El significado de estos botones, que aparece al colocar el cursor sobre cada uno de ellos, es el siguiente:

 ***Set/Clear Breakpoint***. Coloca o borra un ***breakpoint*** en la línea en que está el cursor.

 ***Clear All Breakpoints***. Elimina todos los ***breakpoints*** que haya en el fichero.

 ***Step In***. Avanzar un paso, y si en ese paso hay una llamada a una función de usuario, entra en dicha función.

 ***Single Step***. Avanzar un paso sin entrar en las funciones de usuario que se llamen en esa línea.

 ***Continue***. Continuar la ejecución hasta el siguiente ***breakpoint***.

 ***Quit Debugging***. Terminar la ejecución del ***Debugger***.

Stack. En esta lista desplegable se puede elegir el *contexto*, es decir el ***espacio de trabajo*** o el ámbito de las variables que se quieren examinar. Ya se verá que el espacio

---

[4] Las seis sentencias de ***prueba01.m*** son las siguientes (reagrupadas en dos líneas):

```
A=randn(3,3); B=A'; C=inv(A); D=C*A;
disp('Hola!'); disp('Ya he terminado');
```

de trabajo base (el de las variables creadas desde la línea de comandos) y el espacio de trabajo de cada una de las funciones son diferentes.

El ***Debugger*** es un programa que hay que conocer muy bien, pues es enormemente útil para detectar y corregir errores. Es también enormemente útil para aprender métodos numéricos y técnicas de programación. Para aprender a manejar el ***Debugger*** lo mejor es practicar.

Cuando se está ejecutando un programa con el ***Debugger*** en cualquier momento se puede ir a la línea de comandos de MATLAB y teclear una expresión para ver su resultado. También se puede seleccionar con el ratón una sub-expresión en cualquier línea vista en el ***Editor/Debugger***, clicar con el botón derecho y en el menú contextual que se abre elegir ***Evaluate Selection***. El resultado de evaluar esa sub-expresión aparece en la línea de comandos de MATLAB.

Ya en las versiones anteriores MATLAB disponía de un ***Debugger alfanumérico*** que se utilizaba desde la línea de comandos y en el que está basado el nuevo ***Debugger gráfico*** del que se ha hablado anteriormente. De hecho, al realizar operaciones con el *Debugger gráfico* van apareciendo las correspondientes instrucciones en la línea de comandos de MATLAB. Para más información sobre los comandos del ***Debugger alfanumérico***, buscar en la sección *"Language Constructs and Debugging"* en *"MATLAB Functions by Subject"* del ***Help Desk***.

1.4.4. *WORKSPACE BROWSER*: EL ESPACIO DE TRABAJO DE MATLAB

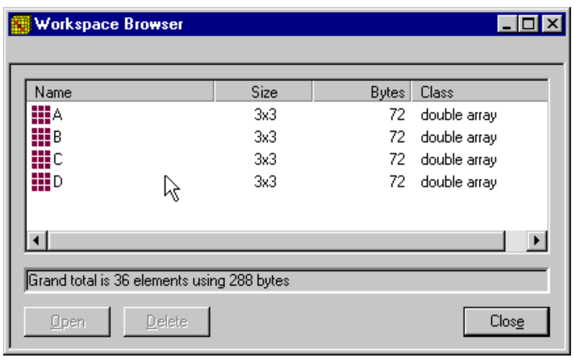
El espacio de trabajo de MATLAB (***Workspace***) es el conjunto de variables y de funciones de usuario que en un determinado momento están definidas en la memoria del programa. Para obtener información sobre el ***Workspace*** se pueden utilizar los comandos ***who*** y ***whos***. El segundo proporciona una información más detallada que el primero. Por ejemplo, después de ejecutar el fichero de comandos ***Prueba01.m***, la salida del comando ***whos*** es la siguiente:

```
» whos
  Name       Size         Bytes  Class

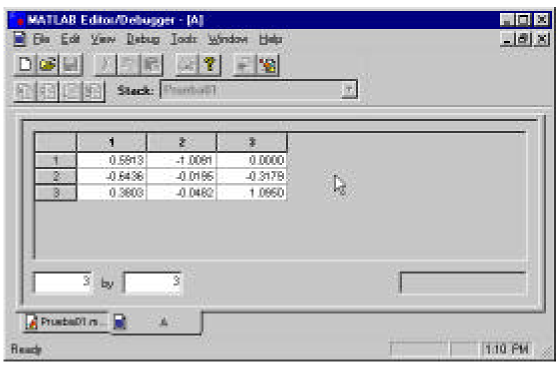
  A          3x3             72  double array
  B          3x3             72  double array
  C          3x3             72  double array
  D          3x3             72  double array

Grand total is 36 elements using 288 bytes
```

Éstas son las variables del ***espacio de trabajo base*** (el de la línea de comandos de MATLAB). Más adelante se verá que cada función tiene su propio espacio de trabajo, con variables cuyos nombres no interfieren con las variables de los otros espacios de trabajo.



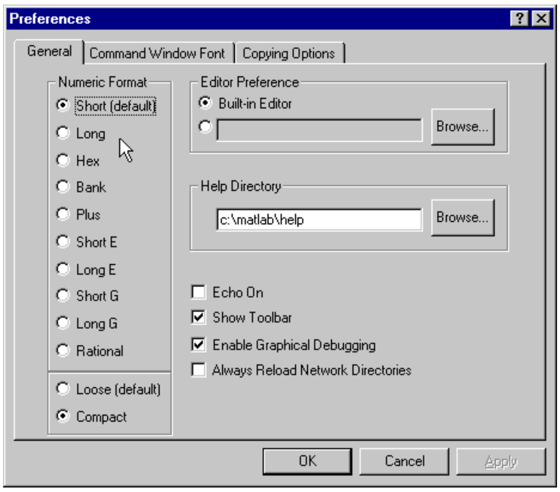
a) Aspecto inicial del ***Workspace Browser***.



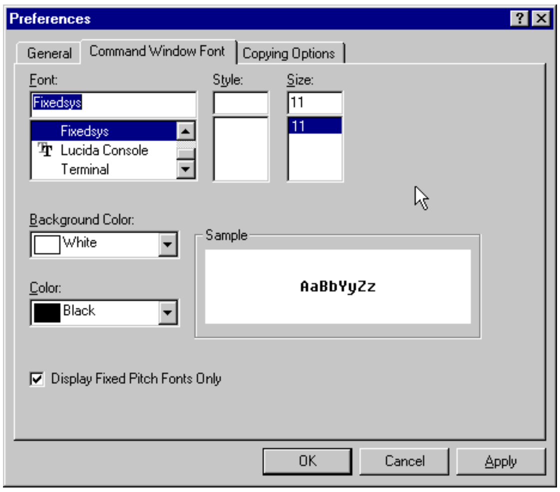
b) Visualización de la matriz **A**.

Figura 9. El ***Workspace Browser*** de MATLAB.

Se puede obtener de modo gráfico una información análoga con el ***Workspace Browser***, que se activa con el comando ***Show Workspace*** del menú ***File*** de MATLAB, o clicando en el botón correspondiente de la barra de herramientas ( ). La Figura 9a muestra el aspecto inicial del ***Workspace Browser*** cuando se abre. Haciendo doble clic sobre la matriz **A** aparece una nueva pestaña en la ventana del ***Editor&Debugger*** en la que se muestran y pueden ser modificados los elementos de dicha matriz (ver Figura 9b).



a) Opciones de tipo ***General***.



b) Opciones de ***Font*** para la ventana de comandos.

Figura 10. Comando ***Preferences*** del menú ***File***.

1.5. Control de los formatos de salida y de otras opciones de MATLAB

Los formatos de salida en la ventana principal de MATLAB se pueden controlar fácilmente a partir del cuadro de diálogo que se abre con el comando ***Preferences*** del menú ***File***. En la Figura 10a y en la Figura 10b se ven dos de los tres cuadros de diálogo a los que se accede al elegir ese comando.

El cuadro de diálogo de la Figura 10a permite elegir un editor de programas distinto del que trae MATLAB, así como elegir el directorio donde están los ficheros ***Help***. Respecto a los formatos numéricos con que MATLAB muestra los resultados (recuérdese que siempre calcula con la máxima precisión), se pueden activar las mismas posibilidades por medio de comandos tecleados en la línea de comandos de MATLAB. Los más importantes de estos comandos son los siguientes:

format short	coma fija con 4 decimales (***defecto***)
format long	coma fija con 15 decimales
format hex	cifras hexadecimales
format bank	números con dos cifras decimales
format short e	notación científica con 4 decimales
format short g	notación científica o decimal, dependiendo del valor
format long e	notación científica con 15 decimales
format long g	notación científica o decimal, dependiendo del valor
format loose	introduce algunas líneas en blanco en la salida (defecto)
format compact	elimina las líneas en blanco citadas (***opción recomendada***)
format rat	expresa los números racionales como cocientes de enteros

MATLAB aplica un ***factor de escala general*** a las matrices cuando los elementos más grandes o más pequeños son superiores o inferiores respectivamente a una determinada cantidad

(10^3 y 10^{-3}). Hay que añadir que MATLAB trata de mantener el formato de los números que han sido definidos como enteros (sin punto decimal). Si se elige la opción ***format rat*** el programa trata de expresar los números racionales como cocientes de enteros.

El cuadro de diálogo ***Command Window Font*** de la Figura 10b ofrece la posibilidad de elegir el tipo de letra –así como el tamaño y el color, tanto de las letras como del fondo– con la que se escribe en la ventana de comandos de MATLAB. Es mejor utilizar tipos de letra de tamaño constante (***Fixedsys*** o ***Courier New***), para que la salida se alinee bien en la pantalla.

1.6. Guardar variables y estados de una sesión: Comandos *save* y *load*

En muchas ocasiones puede resultar interesante interrumpir el trabajo con MATLAB y poderlo recuperar más tarde en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.). Hay que tener en cuenta que al salir del programa todo el contenido de la memoria se borra automáticamente.

Para guardar el estado de una sesión de trabajo en el ***directorio actual*** existe el comando ***save***. Si se teclea:

```
» save
```

antes de abandonar el programa, se crea un fichero binario llamado ***matlab.mat*** (o ***matlab***) con el estado de la sesión (excepto los gráficos, que por ocupar mucha memoria hay que guardar aparte). Dicho estado puede recuperarse la siguiente vez que se arranque el programa con el comando:

```
» load
```

Esta es la forma más básica de los comandos ***save*** y ***load***. Se pueden guardar también matrices y vectores de forma selectiva y en ficheros con nombre especificado por el usuario. Por ejemplo, el comando (sin comas entre los nombres de variables):

```
» save filename A x y
```

guarda las variables **A**, **x** e **y** en un fichero binario llamado ***filename.mat*** (o ***filename***). Para recuperarlas en otra sesión basta teclear:

```
» load filename
```

Si no se indica ningún nombre de variable, se guardan todas las variables creadas en esa sesión.

El comando ***save*** permite guardar el estado de la sesión en formato ASCII utilizándolo de la siguiente forma (lo que va detrás del carácter (%) es un comentario que es ignorado por MATLAB):

```
» save -ascii                % almacena  8 cifras decimales
» save -ascii -double        % almacena 16 cifras decimales
» save -ascii -double -tab   % almacena 16 cifras separadas por tabs
```

aunque en formato ASCII sólo se guardan los valores y no otra información tal como los nombres de las matrices y/o vectores. Cuando se recuperan estos ficheros con ***load -ascii*** toda la información se guarda en una única matriz con el nombre del fichero. Esto produce un error cuando no todas las filas tienen el mismo número de elementos.

Con la opción ***-append*** en el comando ***save*** la información se guarda a continuación de lo que hubiera en el fichero. Es posible también almacenar con el formato binario de la versión 4.* de MATLAB utilizando la opción ***-v4***.

El comando ***load*** admite las opciones ***-ascii*** y ***-mat***, para obligarle a leer en formato ASCII o binario, respectivamente.

1.7. Guardar sesión y copiar salidas: Comando *diary*

Los comandos ***save*** y ***load*** crean ficheros binarios o ASCII con el estado de la sesión. Existe otra forma más sencilla de almacenar en un fichero un texto que describa lo que el programa va haciendo (la entrada y salida de los comandos utilizados). Esto se hace con el comando ***diary*** en la forma siguiente:

```
» diary filename.txt
...
» diary off
...
» diary on
...
```

El comando ***diary off*** suspende la ejecución de ***diary*** y ***diary on*** la reanuda. El simple comando ***diary*** pasa de ***on*** a ***off*** y viceversa. Para poder acceder al fichero ***filename.txt*** con ***Notepad*** es necesario que ***diary*** esté en ***off***.

Si no se incluye el nombre del fichero se utiliza por defecto un fichero llamado ***diary***.

1.8. Líneas de comentarios

Ya se ha indicado que para MATLAB el carácter ***tanto por ciento*** (%) indica comienzo de comentario. Cuando aparece en una línea de comandos, el programa supone que todo lo que va desde ese carácter hasta el fin de la línea es un comentario.

Más adelante se verá que los comentarios de los ficheros ****.m*** tienen algunas peculiaridades importantes, pues pueden servir para definir ***help***'s personalizados de las funciones que el usuario vaya creando.

1.9. Medida de tiempos y de esfuerzo de cálculo

MATLAB dispone de funciones que permiten calcular el tiempo empleado en las operaciones matemáticas realizadas. Algunas de estas funciones son las siguientes:

cputime	devuelve el tiempo de CPU (con precisión de centésimas de segundo) desde que el programa arrancó. Llamando antes y después de realizar una operación y restando los valores devueltos, se puede saber el tiempo de CPU empleado en esa operación. Este tiempo sigue corriendo aunque MATLAB esté inactivo.
etime(t2, t1)	tiempo transcurrido entre los vectores **t1** y **t2** (¡atención al orden!), obtenidos como respuesta al comando ***clock***.
tic *ops* toc	imprime el tiempo en segundos requerido por ***ops***. El comando ***tic*** pone el reloj a cero y ***toc*** obtiene el tiempo transcurrido.

Otras funciones permiten calcular el número de operaciones de coma flotante realizadas, como las siguientes:

flops(0)	inicializa a cero el contador de número de operaciones aritméticas de punto flotante (*flops*)
flops	devuelve el número de *flops* realizados hasta ese momento

Por ejemplo, el siguiente código mide de varias formas el tiempo necesario para resolver un sistema de 500 ecuaciones con 500 incógnitas. Téngase en cuenta que los tiempos pequeños (del orden de las décimas o centésimas de segundo), no se pueden medir con gran precisión.

```
» A=rand(500); b=rand(100,1); x=zeros(500,1);
» tiempo=clock; x=A\b; tiempo=etime(clock, tiempo)
» time=cputime; x=A\b; time=cputime-time
» tic; x=A\b; toc
```

donde se han puesto varias sentencias en la misma línea para que se ejecuten todas sin tiempos muertos al pulsar ***intro***. Esto es especialmente importante en la línea de comandos en la que se quiere medir los tiempos. Todas las sentencias de cálculos matriciales van seguidas de punto y coma (;) con objeto de evitar la impresión de resultados.

2. OPERACIONES CON MATRICES Y VECTORES

Ya se ha comentado que MATLAB es fundamentalmente un programa para cálculo matricial. Inicialmente se utilizará MATLAB como ***programa interactivo***, en el que se irán definiendo las matrices, los vectores y las expresiones que los combinan y obteniendo los resultados sobre la marcha. Si estos resultados son asignados a otras variables podrán ser utilizados posteriormente en otras expresiones. En este sentido MATLAB sería como una potente calculadora matricial (en realidad es esto y mucho más...).

Antes de tratar de hacer cálculos complicados, la primera tarea será aprender a introducir matrices y vectores desde el teclado. Más adelante se verán otras formas más potentes de definir matrices y vectores.

2.1. Definición de matrices desde teclado

Como en casi todos los lenguajes de programación, en MATLAB las matrices y vectores son ***variables*** que tienen ***nombres***. Ya se verá luego con más detalle las reglas que deben cumplir estos nombres. Por el momento se sugiere que se utilicen letras mayúsculas para matrices y minúsculas para vectores y escalares (MATLAB no exige esto, pero puede resultar útil).

Para definir una matriz *no hace falta establecer de antemano su tamaño* (de hecho, se puede definir un tamaño y cambiarlo posteriormente). MATLAB determina el número de filas y de columnas en función del número de elementos que se proporcionan (o se utilizan). ***Las matrices se definen por filas***; los elementos de una misma fila están separados por ***blancos*** o ***comas***, mientras que las filas están separadas por pulsaciones ***intro*** o por caracteres ***punto y coma*** (;). Por ejemplo, el siguiente comando define una matriz **A** de dimensión (3x3):

```
» A=[1 2 3; 4 5 6; 7 8 9]
```

La respuesta del programa es la siguiente:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

A partir de este momento la matriz **A** está disponible para hacer cualquier tipo de operación con ella (además de valores numéricos, en la definición de una matriz o vector se pueden utilizar expresiones y funciones matemáticas). Por ejemplo, una sencilla operación con **A** es hallar su ***matriz traspuesta***. En MATLAB el apóstrofo (') es el símbolo de *trasposición matricial*. Para calcular **A'** (traspuesta de **A**) basta teclear lo siguiente (se añade a continuación la respuesta del programa):

```
» A'
ans =
     1     4     7
     2     5     8
     3     6     9
```

Como el resultado de la operación no ha sido asignado a ninguna otra matriz, MATLAB utiliza un nombre de variable por defecto (***ans***, de *answer*), que contiene el resultado de la última operación. La variable ***ans*** puede ser utilizada como operando en la siguiente expresión que se introduzca. También podría haberse asignado el resultado a otra matriz llamada **B**:

```
» B=A'
B =
     1     4     7
     2     5     8
     3     6     9
```

Ahora ya están definidas las matrices **A** y **B**, y es posible seguir operando con ellas. Por ejemplo, se puede hacer el producto **B*A** (deberá resultar una matriz simétrica):

```
» B*A
ans =
    66    78    90
    78    93   108
    90   108   126
```

En MATLAB se accede a los elementos de un vector poniendo el índice entre paréntesis (por ejemplo *x(3)* ó *x(i)*). Los elementos de las matrices se acceden poniendo los dos índices entre paréntesis, separados por una coma (por ejemplo *A(1,2)* ó *A(i,j)*). Las matrices ***se almacenan por columnas*** (aunque ***se introduzcan por filas***, como se ha dicho antes), y teniendo en cuenta esto puede accederse a cualquier elemento de una matriz con un sólo subíndice. Por ejemplo, si **A** es una matriz (3x3) se obtiene el mismo valor escribiendo *A(1,2)* que escribiendo *A(4)*.

Invertir una matriz es casi tan fácil como trasponerla. A continuación se va a definir una nueva matriz **A** -no singular- en la forma:

```
» A=[1 4 -3; 2 1 5; -2 5 3]
A =
     1     4    -3
     2     1     5
    -2     5     3
```

Ahora se va a calcular la inversa de **A** y el resultado se asignará a **B**. Para ello basta hacer uso de la función ***inv( )*** (la precisión o número de cifras con que se muestra el resultado se puede cambiar con el menú ***File/Preferences/General***):

```
B=inv(A)
B =
    0.1803    0.2213   -0.1885
    0.1311    0.0246    0.0902
   -0.0984    0.1066    0.0574
```

Para comprobar que este resultado es correcto basta pre-multiplicar **A** por **B**;

```
» B*A
ans =
    1.0000    0.0000    0.0000
    0.0000    1.0000    0.0000
    0.0000    0.0000    1.0000
```

De forma análoga a las matrices, es posible definir un ***vector fila*** **x** en la forma siguiente (si los tres números están separados por *blancos* o *comas*, el resultado será un vector fila):

```
» x=[10 20 30]   % vector fila
x =
    10    20    30
```

MATLAB considera ***comentarios*** todo lo que va desde el ***carácter tanto por ciento*** (***%***) hasta el final de la línea.

Por el contrario, si los números están separados por *intros* o *puntos y coma* (;) se obtendrá un ***vector columna***:

```
» y=[11; 12; 13]   % vector columna
y =
    11
    12
    13
```

MATLAB tiene en cuenta la diferencia entre vectores fila y vectores columna. Por ejemplo, si se intenta sumar los vectores **x** e **y** se obtendrá el siguiente mensaje de error:

```
» x+y
??? Error using ==> +
Matrix dimensions must agree.
```

Estas dificultades desaparecen si se suma **x** con el vector traspuesto de **y**:

```
» x+y'
ans =
    21    32    43
```

Aunque ya se ha visto en los ejemplos anteriores el estilo sencillo e intuitivo con el que MATLAB opera con matrices y vectores, a continuación se va a estudiar este tema con un poco más de detenimiento.

2.2. Operaciones con matrices

MATLAB puede operar con matrices por medio de ***operadores*** y por medio de ***funciones***. Se han visto ya los operadores *suma* (+), *producto* (*) y *traspuesta* ('), así como la función *invertir* ***inv( )***. Los operadores matriciales de MATLAB son los siguientes:

+	adición o suma
–	sustracción o resta
*	multiplicación
'	traspuesta
^	potenciación
\	división-izquierda
/	división-derecha
.*	producto elemento a elemento
./ y .\	división elemento a elemento
.^	elevar a una potencia elemento a elemento

Estos operadores se aplican también a las variables o valores escalares, aunque con algunas diferencias[5]. Todos estos operadores son coherentes con las correspondientes operaciones matriciales: no se puede por ejemplo sumar matrices que no sean del mismo tamaño. Si los operadores no se usan de modo correcto se obtiene un mensaje de error.

Los operadores anteriores se pueden aplicar también de modo ***mixto***, es decir con un operando escalar y otro matricial. En este caso la operación con el escalar se aplica a cada uno de los elementos de la matriz. Considérese el siguiente ejemplo:

```
» A=[1 2; 3 4]
A =
     1     2
     3     4
```

5 En términos de C++ se podría decir que son operadores *sobrecargados*, es decir, con varios significados distintos dependiendo del contexto, es decir, de sus operandos.

```
» A*2
ans =
     2     4
     6     8
» A-4
ans =
    -3    -2
    -1     0
```

Los ***operadores de división*** requieren una cierta explicación adicional. Considérese el siguiente sistema de ecuaciones lineales,

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

en donde **x** y **b** son vectores columna, y **A** una matriz cuadrada invertible. La resolución de este sistema de ecuaciones se puede escribir en las 2 formas siguientes (¡Atención a la 2ª forma, basada en la *barra invertida* (\), que puede resultar un poco extraña!):

$$\mathbf{x} = \mathrm{inv}(\mathbf{A})*\mathbf{b} \tag{2a}$$

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{b} \tag{2b}$$

Así pues, el operador *división-izquierda* por una matriz (barra invertida \) equivale a premultiplicar por la inversa de esa matriz. En realidad este operador es más general y más inteligente de lo que aparece en el ejemplo anterior: el operador *división-izquierda* es aplicable aunque la matriz no tenga inversa e incluso no sea cuadrada, en cuyo caso la solución que se obtiene (por lo general) es la que proporciona el *método de los mínimos cuadrados*. Cuando la matriz es triangular o simétrica aprovecha esta circunstancia para reducir el número de operaciones aritméticas. En algunos casos se obtiene una solución de mínima con no más de ***r*** elementos distintos de cero, siendo ***r*** el rango de la matriz. Esto puede estar basado en que la matriz se reduce a forma de escalón y se resuelve el sistema dando valor cero a las variables independientes. Por ejemplo, considérese el siguiente ejemplo de matriz (1x2) que conduce a un sistema de infinitas soluciones:

```
» A=[1 2], b=[2]
A =
     1     2
b =
     2
» x=A\b
x =
     0
     1
```

que es la solución obtenida dando valor cero a la variable independiente x(1). Por otra parte, en el caso de un sistema de ecuaciones ***redundante*** (o *sobre-determinado*) el resultado de MATLAB es el punto más "cercano" -en el sentido de mínima norma del error- a las ecuaciones dadas (aunque no cumpla exactamente ninguna de ellas). Véase el siguiente ejemplo de tres ecuaciones formadas por una recta que no pasa por el origen y los dos ejes de coordenadas:

```
» A=[1 2; 1 0; 0 1], b=[2 0 0]'
A =
     1     2
     1     0
     0     1
b =
     2
     0
     0
```

```
» x=A\b, resto=A*x-b
x =
    0.3333
    0.6667
resto =
   -0.3333
    0.3333
    0.6667
```

Aunque no es una forma demasiado habitual, también se puede escribir un sistema de ecuaciones lineales en la forma correspondiente a la traspuesta de la ecuación (1):

$$\mathbf{yB} = \mathbf{c} \tag{3}$$

donde **y** y **c** son vectores fila (**c** conocido). Si la matriz **B** es cuadrada e invertible, la solución de este sistema se puede escribir en las formas siguientes:

$$\mathbf{y} = \mathbf{c}*\text{inv}(\mathbf{B}) \tag{4a}$$

$$\mathbf{y} = \mathbf{c}/\mathbf{B} \tag{4b}$$

En este caso, el operador *división-derecha* por una matriz (/) equivale a postmultiplicar por la inversa de la matriz. Si se traspone la ecuación (3) y se halla la solución aplicando el operador *división-izquierda* se obtiene:

$$\mathbf{y}' = (\mathbf{B}')\backslash\mathbf{c}' \tag{5}$$

Comparando las expresiones (4b) y (5) se obtiene la relación entre los operadores *división-izquierda* y *división-derecha* (MATLAB sólo tiene implementado el operador *división-izquierda*):

$$\mathbf{c}/\mathbf{B} = ((\mathbf{B}')\backslash\mathbf{c}')' \tag{6}$$

En MATLAB existe también la posibilidad de aplicar ***elemento a elemento*** los operadores matriciales (*, ^, \ y /). Para ello basta precederlos por un punto (.). Por ejemplo:

```
» [1 2 3 4]^2
??? Error using ==> ^
Matrix must be square.

» [1 2 3 4].^2
ans =
     1     4     9    16

» [1 2 3 4]*[1 -1 1 -1]
??? Error using ==> *
Inner matrix dimensions must agree.

» [1 2 3 4].*[1 -1 1 -1]
ans =
     1    -2     3    -4
```

2.3. Tipos de datos

Ya se ha dicho que MATLAB es un programa preparado para trabajar con vectores y matrices. Como caso particular también trabaja con variables escalares (matrices de dimensión 1). MATLAB trabaja siempre en *doble precisión*, es decir guardando cada dato en 8 bytes, con unas 15 cifras decimales exactas. Ya se verá más adelante que también puede trabajar con cadenas de caracteres (*strings*) y, desde la versión 5.0, también con otros tipos de datos: ***Matrices de más dos dimensiones***, ***matrices dispersas***, ***vectores y matrices de celdas***, ***estructuras*** y ***clases y objetos***. Algunos de estos tipos de datos más avanzados se verán en la última parte de este manual.

2.3.1. NÚMEROS REALES DE DOBLE PRECISIÓN

Los elementos constitutivos de vectores y matrices son números reales almacenados en 8 bytes (53 bits para la mantisa y 11 para el exponente de 2; entre 15 y 16 cifras decimales equivalentes). Es importante saber cómo trabaja MATLAB con estos números y los casos especiales que presentan.

MATLAB mantiene una forma especial para los *números muy grandes* (más grandes que los que es capaz de representar), que son considerados como ***infinito***. Por ejemplo, obsérvese cómo responde el programa al ejecutar el siguiente comando:

```
» 1.0/0.0
Warning: Divide by zero
ans =
   Inf
```

Así pues, para MATLAB el ***infinito*** se representa como ***inf*** ó ***Inf***. MATLAB tiene también una representación especial para los resultados que no están definidos como números. Por ejemplo, ejecútense los siguientes comandos y obsérvense las respuestas obtenidas:

```
» 0/0
Warning: Divide by zero
ans =
   NaN
» inf/inf
ans =
   NaN
```

En ambos casos la respuesta es ***NaN***, que es la abreviatura de ***Not a Number***. Este tipo de respuesta, así como la de ***Inf***, son enormemente importantes en MATLAB, pues permiten controlar la fiabilidad de los resultados de los cálculos matriciales. Los ***NaN*** se propagan al realizar con ellos cualquier operación aritmética, en el sentido de que, por ejemplo, cualquier número sumado a un ***NaN*** da otro ***NaN***. MATLAB tiene esto en cuenta. Algo parecido sucede con los ***Inf***.

MATLAB dispone de tres funciones útiles relacionadas con las operaciones de coma flotante. Estas funciones, que no tienen argumentos, son las siguientes:

eps devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. Da una idea de la precisión o número de cifras almacenadas. En un PC, ***eps*** vale 2.2204e-016.

realmin devuelve el número más pequeño con que se puede trabajar (2.2251e-308)

realmax devuelve el número más grande con que se puede trabajar (1.7977e+308)

2.3.2. NÚMEROS COMPLEJOS: FUNCIÓN *COMPLEX*

En muchos cálculos matriciales los datos y/o los resultados no son reales sino ***complejos***, con ***parte real*** y ***parte imaginaria***. MATLAB trabaja sin ninguna dificultad con números complejos. Para ver como se representan por defecto los números complejos, ejecútense los siguientes comandos:

```
» a=sqrt(-4)
a =
        0 + 2.0000i
» 3 + 4j
ans =
   3.0000 + 4.0000i
```

En la entrada de datos de MATLAB se pueden utilizar indistintamente la **i** y la **j** para representar el *número imaginario unidad* (en la salida, sin embargo, puede verse que siempre aparece la **i**). Si la **i** o la **j** no están definidas como variables, puede intercalarse el signo (*). Esto no

es posible en el caso de que sí estén definidas, porque entonces se utiliza el valor de la variable. En general, cuando se está trabajando con números complejos, conviene no utilizar la **i** como variable ordinaria, pues puede dar lugar a errores y confusiones. Por ejemplo, obsérvense los siguientes resultados:

```
» i=2
i =
     2
» 2+3i
ans =
   2.0000 + 3.0000i
» 2+3*i
ans =
     8
» 2+3*j
ans =
   2.0000 + 3.0000i
```

Cuando **i** y **j** son variables utilizadas para otras finalidades, como *unidad imaginaria* puede utilizarse también la función ***sqrt(-1)***, o una variable a la que se haya asignado el resultado de esta función.

La asignación de *valores complejos* a vectores y matrices desde teclado puede hacerse de las dos formas, que se muestran en el ejemplo siguiente (conviene hacer antes ***clear i***, para que **i** no esté definida como variable. Este comando se estudiará más adelante):

```
» A = [1+2i 2+3i; -1+i 2-3i]
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
» A = [1 2; -1 2] + [2 3; 1 -3]*I   % En este caso el * es necesario
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
```

Puede verse que es posible definir las partes reales e imaginarias por separado. En este caso sí es necesario utilizar el operador (*), según se muestra en el ejemplo anterior.

MATLAB dispone asimismo de la función ***complex***, que crea un número complejo a partir de dos argumentos que representan la parte real e imaginaria, como en el ejemplo siguiente:

```
» complex(1,2)
ans =
   1.0000 + 2.0000i
```

Es importante advertir que el *operador de matriz traspuesta* ('), aplicado a matrices complejas, produce la matriz conjugada y traspuesta. Existe una función que permite hallar simplemente la matriz conjugada (***conj()***) y el operador punto y apóstrofo (**.'**) que calcula simplemente la matriz traspuesta.

2.3.3. CADENAS DE CARACTERES

MATLAB puede definir variables que contengan cadenas de caracteres. En MATLAB las cadenas de texto van entre apóstrofos o comillas simples (Nótese que en C van entre comillas dobles: "cadena"). Por ejemplo, en MATLAB:

```
s = 'cadena de caracteres'
```

Las cadenas de texto tienen su más clara utilidad en temas que se verán más adelante y por eso se difiere hasta entonces una explicación más detallada.

2.4. Variables y expresiones matriciales

Ya han aparecido algunos ejemplos de ***variables*** y ***expresiones*** matriciales. Ahora se va a tratar de generalizar un poco lo visto hasta ahora.

Una ***variable*** es un nombre que se da a una entidad numérica, que puede ser una matriz, un vector o un escalar. El valor de esa variable, e incluso el tipo de entidad numérica que representa, puede cambiar a lo largo de una sesión de MATLAB o a lo largo de la ejecución de un programa. La forma más normal de cambiar el valor de una variable es colocándola a la izquierda del ***operador de asignación*** (=).

Una expresión de MATLAB puede tener las dos formas siguientes: primero, asignando su resultado a una variable,

```
variable = expresión
```

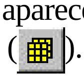
y segundo evaluando simplemente el resultado del siguiente modo,

```
expresión
```

en cuyo caso el resultado se asigna automáticamente a una variable interna de MATLAB llamada ***ans*** (de *answer*) que almacena el último resultado obtenido. Se considera por defecto que una expresión termina cuando se pulsa ***intro***. Si se desea que una expresión continúe en la línea siguiente, hay que introducir ***tres puntos*** (...) antes de pulsar ***intro***. También se pueden incluir varias expresiones en una misma línea separándolas por *comas* (,) o *puntos y comas* (;).

Si una expresión ***termina en punto y coma*** (;) su resultado se calcula, pero no se escribe en pantalla. Esta posibilidad es muy interesante, tanto para evitar la escritura de resultados intermedios, como para evitar la impresión de grandes cantidades de números cuando se trabaja con matrices de gran tamaño.

A semejanza de C, ***MATLAB distingue entre mayúsculas y minúsculas*** en los nombres de variables. Los ***nombres de variables*** deben empezar siempre por una letra y pueden constar de hasta 31 letras y números. El carácter guión bajo (_) se considera como una letra. A diferencia del lenguaje C, no hace falta declarar las variables que se vayan a utilizar. Esto hace que se deba tener especial cuidado con no utilizar nombres erróneos en las variables, porque no se recibirá ningún aviso del ordenador.

Cuando se quiere tener una *relación de las variables* que se han utilizado en una sesión de trabajo se puede utilizar el comando ***who***. Existe otro comando llamado ***whos*** que proporciona además información sobre el tamaño, la cantidad de memoria ocupada y el carácter real o complejo de cada variable. Se sugiere utilizar de vez en cuando estos comandos en la sesión de MATLAB que se tiene abierta. Esta misma información se puede obtener gráficamente con el ***Workspace Browser***, que aparece con el comando ***Show Workspace*** del menú ***File*** o clicando en el icono correspondiente ().

El comando ***clear*** tiene varias formas posibles:

clear	sin argumentos, ***clear*** elimina todas las variables creadas previamente (excepto las variables globales).
clear A, b	borra las variables indicadas.
clear global	borra las variables globales.
clear functions	borra las funciones.
clear all	borra todas las variables, incluyendo las globales, y las funciones.

2.5. Otras formas de definir matrices

MATLAB dispone de varias formas de definir matrices. El introducirlas por teclado sólo es práctico en casos de pequeño tamaño y cuando no hay que repetir esa operación muchas veces. Recuérdese que en MATLAB no hace falta definir el tamaño de una matriz. Las matrices toman tamaño al ser definidas y este tamaño puede ser modificado por el usuario mediante adición y/o borrado de filas y columnas. A continuación se van a ver otras formas más potentes y generales de definir y/o modificar matrices.

2.5.1. TIPOS DE MATRICES PREDEFINIDOS

Existen en MATLAB varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. Algunas de estas funciones son las siguientes:

eye(4)	forma la matriz unidad de tamaño (4x4)
zeros(3,5)	forma una matriz de *ceros* de tamaño (3x5)
zeros(4)	ídem de tamaño (4x4)
ones(3)	forma una matriz de *unos* de tamaño (3x3)
ones(2,4)	idem de tamaño (2x4)
linspace(x1,x2,n)	genera un vector con **n** valores igualmente espaciados entre **x1** y **x2**
logspace(d1,d2,n)	genera un vector con **n** valores espaciados logarítmicamente entre 10^d1 y 10^d2. Si d2 es **pi**[6], los puntos se generan entre 10^d1 y **pi**
rand(3)	forma una matriz de números aleatorios entre 0 y 1, con distribución uniforme, de tamaño (3x3)
rand(2,5)	idem de tamaño (2x5)
randn(4)	forma una matriz de números aleatorios de tamaño (4x4), con distribución normal, de valor medio 0 y varianza 1.
magic(4)	crea una matriz (4x4) con los números 1, 2, ... 4*4, con la propiedad de que todas las filas y columnas suman lo mismo
hilb(5)	crea una matriz de Hilbert de tamaño (5x5). La matriz de Hilbert es una matriz cuyos elementos (i,j) responden a la expresión (1/(i+j-1)). Esta es una matriz especialmente difícil de manejar por los grandes errores numéricos a los que conduce
invhilb(5)	crea directamente la inversa de la matriz de Hilbert
kron(x,y)	produce una matriz con todos los productos de los elementos del vector **x** por los elementos del vector **y**. Equivalente a **x**'***y**, donde **x** e **y** son vectores fila
compan(pol)	construye una matriz cuyo polinomio característico tiene como coeficientes los elementos del vector **pol** (ordenados de mayor grado a menor)
vander(v)	construye la matriz de ***Vandermonde*** a partir del vector **v** (las columnas son las potencias de los elementos de dicho vector)

Existen otras funciones para crear matrices de tipos particulares. Con ***Help/Help Window*** se puede obtener información sobre todas las funciones disponibles en MATLAB, que aparecen

[6] **pi** es una variable predefinida en MATLAB, que como es fácil suponer representa el número π.

agrupadas por directorios. En ***matlab\elmat*** aparecen la mayor parte de las funciones estudiadas en este apartado.

2.5.2. FORMACIÓN DE UNA MATRIZ A PARTIR DE OTRAS

MATLAB ofrece también la posibilidad de crear una matriz a partir de matrices previas ya definidas, por varios posibles caminos:

– recibiendo alguna de sus propiedades (como por ejemplo el tamaño),

– por composición de varias submatrices más pequeñas,

– modificándola de alguna forma.

A continuación se describen algunas de las funciones que crean una nueva matriz a partir de otra o de otras, comenzando por dos funciones auxiliares:

[m,n]=size(A)	devuelve el número de filas y de columnas de la matriz **A**. Si la matriz es cuadrada basta recoger el primer valor de retorno
n=length(x)	calcula el número de elementos de un vector **x**
zeros(size(A))	forma una matriz de *ceros* del mismo tamaño que una matriz **A** previamente creada
ones(size(A))	ídem con *unos*
A=diag(x)	forma una matriz diagonal **A** cuyos elementos diagonales son los elementos de un vector ya existente **x**
x=diag(A)	forma un vector **x** a partir de los elementos de la diagonal de una matriz ya existente **A**
diag(diag(A))	crea una matriz diagonal a partir de la diagonal de la matriz **A**
blkdiag(A,B)	crea una matriz diagonal de submatrices a partir de las matrices que se le pasan como argumentos
triu(A)	forma una matriz triangular superior a partir de una matriz **A** (no tiene por qué ser cuadrada). Con un segundo argumento puede controlarse que se mantengan o eliminen más diagonales por encima o debajo de la diagonal principal.
tril(A)	ídem con una matriz triangular inferior
rot90(A,k)	Gira k*90 grados la matriz rectangular **A** en sentido antihorario. k es un entero que puede ser negativo. Si se omite, se supone k=1
flipud(A)	halla la matriz simétrica de **A** respecto de un eje horizontal
fliplr(A)	halla la matriz simétrica de **A** respecto de un eje vertical
reshape(A,m,n)	Cambia el tamaño de la matriz **A** devolviendo una matriz de tamaño $m \times n$ cuyas columnas se obtienen a partir de un vector formado por las columnas de **A** puestas una a continuación de otra. Si la matriz **A** tiene menos de $m \times n$ elementos se produce un error.

Un caso especialmente interesante es el de crear una nueva matriz ***componiendo como submatrices*** otras matrices definidas previamente. A modo de ejemplo, ejecútense las siguientes líneas de comandos y obsérvense los resultados obtenidos:

```
» A=rand(3)
» B=diag(diag(A))
» C=[A, eye(3); zeros(3), B]
```

En el ejemplo anterior, la matriz **C** de tamaño (6x6) se forma por composición de cuatro matrices de tamaño (3x3). Al igual que con simples escalares, las submatrices que forman una fila

se separan con ***blancos*** o ***comas***, mientras que las diferentes filas se separan entre sí con ***intros*** o ***puntos y comas***. Los tamaños de las submatrices deben de ser coherentes.

2.5.3. DIRECCIONAMIENTO DE VECTORES Y MATRICES A PARTIR DE VECTORES

Los elementos de un vector **x** se pueden direccionar a partir de los de otro vector **v**. En este caso, **x**(**v**) equivale al vector x(v(1)), x(v(2)), ... Considérese el siguiente ejemplo:

```
» v=[1 3 4]
v =
     1     3     4
» x=rand(1,6)
x =
    0.5899    0.4987    0.7351    0.9231    0.1449    0.9719
» x(v)
ans =
    0.5899    0.7351    0.9231
```

De forma análoga, los elementos de una matriz **A** pueden direccionarse a partir de los elementos de dos vectores **f** y **c**. Véase por ejemplo:

```
» f=[2 4]; c=[1 2];
» A=magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
» A(f,c)
ans =
     5    11
     4    14
```

El siguiente ejemplo –continuación del anterior– permite comprobar cómo los elementos de una matriz se pueden direccionar con un sólo índice, considerando que las columnas de la matriz están una a continuación de otra formando un vector:

```
» f=[1 3 5 7];
» A(f), A(5), A(6)
ans =
    16     9     2     7
ans =
     2
ans =
    11
```

Más adelante se verá que esta forma de extraer elementos de un vector y/o de una matriz tiene abundantes aplicaciones, por ejemplo la de modificar selectivamente esos elementos.

2.5.4. OPERADOR DOS PUNTOS (:)

Este operador es muy importante en MATLAB y puede usarse de varias formas. Se sugiere al lector que practique mucho sobre los ejemplos contenidos en este apartado, introduciendo todas las modificaciones que se le ocurran y haciendo pruebas abundantes (¡Probar es la mejor forma de aprender!).

Para empezar, defínase un vector **x** con el siguiente comando:

```
» x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
```

En cierta forma se podría decir que el operador (:) representa un ***rango***: en este caso, los números enteros entre el 1 y el 10. Por defecto el incremento es 1, pero este operador puede también utilizarse con otros valores enteros y reales, positivos o negativos. En este caso el incremento va entre el valor inferior y el superior, en las formas que se muestran a continuación:

```
» x=1:2:10
x =
     1     3     5     7     9
» x=1:1.5:10
x =
    1.0000    2.5000    4.0000    5.5000    7.0000    8.5000   10.0000
» x=10:-1:1
x =
    10     9     8     7     6     5     4     3     2     1
```

Puede verse que, por defecto, este operador produce vectores fila. Si se desea obtener un vector columna basta trasponer el resultado. El siguiente ejemplo genera una tabla de funciones *seno* y *coseno*. Ejecútese y obsérvese el resultado (recuérdese que con (;) después de un comando el resultado no aparece en pantalla).

```
» x=[0.0:pi/50:2*pi]';
» y=sin(x); z=cos(x);
» [x y z]
```

El operador dos puntos (:) es aún más útil y potente –y también más complicado– con matrices. A continuación se va a definir una matriz **A** de tamaño 6x6 y después se realizarán diversas operaciones sobre ella con el operador (:).

```
» A=magic(6)
A =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
```

Recuérdese que MATLAB accede a los elementos de una matriz por medio de los índices de fila y de columna encerrados entre paréntesis y separados por una coma. Por ejemplo:

```
» A(2,3)
ans =
     7
```

El siguiente comando extrae los 4 primeros elementos de la 6ª fila:

```
» A(6, 1:4)
ans =
     4    36    29    13
```

Los dos puntos aislados representan "todos los elementos". Por ejemplo, el siguiente comando extrae todos los elementos de la 3ª fila:

```
» A(3, :)
ans =
    31     9     2    22    27    20
```

Para acceder a la última fila o columna puede utilizarse la palabra ***end***, en lugar del número correspondiente. Por ejemplo, para extraer la sexta fila (la última) de la matriz:

```
» A(end, :)
ans =
     4    36    29    13    18    11
```

El siguiente comando extrae todos los elementos de las filas 3, 4 y 5:

```
» A(3:5,:)
ans =
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
```

Se pueden extraer conjuntos disjuntos de filas utilizando *corchetes* [ ]. Por ejemplo, el siguiente comando extrae las filas 1, 2 y 5:

```
» A([1 2 5],:)
ans =
    35     1     6    26    19    24
     3    32     7    21    23    25
    30     5    34    12    14    16
```

En los ejemplos anteriores se han extraído filas y no columnas por motivos del espacio ocupado por el resultado en la hoja de papel. Es evidente que todo lo que se dice para filas vale para columnas y viceversa: basta cambiar el orden de los índices.

El operador dos puntos (:) puede utilizarse en ambos lados del operador (=). Por ejemplo, a continuación se va a definir una matriz identidad **B** de tamaño 6x6 y se van a reemplazar filas de **B** por filas de **A**. Obsérvese que la siguiente secuencia de comandos sustituye las filas 2, 4 y 5 de **B** por las filas 1, 2 y 3 de **A**,

```
» B=eye(size(A));
» B([2 4 5],:)=A(1:3,:)
B =
     1     0     0     0     0     0
    35     1     6    26    19    24
     0     0     1     0     0     0
     3    32     7    21    23    25
    31     9     2    22    27    20
     0     0     0     0     0     1
```

Se pueden realizar operaciones aún más complicadas, tales como la siguiente[7]:

```
» B=eye(size(A));
» B(1:2,:)=[0 1; 1 0]*B(1:2,:)
```

Como nuevo ejemplo, se va a ver la forma de invertir el orden de los elementos de un vector:

```
» x=rand(1,5)
x =
    0.9103    0.7622    0.2625    0.0475    0.7361
» x=x(5:-1:1)
x =
    0.7361    0.0475    0.2625    0.7622    0.9103
```

Obsérvese que por haber utilizado paréntesis –en vez de corchetes– los valores generados por el operador (:) afectan a los índices del vector y no al valor de sus elementos.

Para invertir el orden de las columnas de una matriz se puede hacer lo siguiente:

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
```

[7] Se sustituyen las dos primeras filas de **B** por el producto de dichas filas por una matriz de permutación.

```
» A(:,3:-1:1)
ans =
     6     1     8
     7     5     3
     2     9     4
```

aunque hubiera sido más fácil utilizar la función ***fliplr(A)***, que es específica para ello.

Finalmente, hay que decir que ***A(:)*** representa un vector columna con las columnas de **A** una detrás de otra.

2.5.5. MATRIZ VACÍA A[ ]

Para MATLAB una matriz definida sin ningún elemento entre los corchetes es una matriz que *existe*, pero que está *vacía*, o lo que es lo mismo que tiene *dimensión cero*. Considérense los siguientes ejemplos de aplicación de las matrices vacías:

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
» B=[]
B =
     []
» exist(B)
ans =
     []
» isempty(B)
ans =
     1
» A(:,3)=[]
A =
     8     1
     3     5
     4     9
```

Las funciones ***exist()*** e ***isempty()*** permiten chequear si una variable existe y si está vacía. En el último ejemplo se ha eliminado la 3ª columna de **A** asignándole la matriz vacía.

2.5.6. DEFINICIÓN DE VECTORES Y MATRICES A PARTIR DE UN FICHERO

MATLAB acepta como entrada un fichero ***nombre.m*** (siempre con extensión ***.m***) que contiene instrucciones y/o funciones. Dicho fichero se llama desde la línea de comandos tecleando simplemente su nombre, sin la extensión. A su vez, un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo (funciones recursivas). Las variables definidas dentro de un fichero de comandos ****.m*** que se ejecuta desde la línea de comandos son variables del ***espacio de trabajo base***, esto es, pueden ser accedidas desde fuera de dicho fichero; no sucede lo mismo si el fichero ****.m*** corresponde a una función. Si un fichero de comandos se llama desde una función, las variables que se crean pertenecen al espacio de trabajo de dicha función.

Como ejemplo se puede crear un fichero llamado ***unidad.m*** que construya una matriz unidad de tamaño 3x3 llamada ***U33*** en un directorio llamado ***g:\matlab***. Este fichero deberá contener la línea siguiente:

```
U33=eye(3)
```

Desde MATLAB llámese al comando ***unidad*** y obsérvese el resultado. Entre otras razones, es muy importante utilizar ficheros de comandos para poder utilizar el ***Debugger*** y para evitar teclear muchas veces los mismos datos, sentencias o expresiones.

2.5.7. DEFINICIÓN DE VECTORES Y MATRICES MEDIANTE FUNCIONES Y DECLARACIONES

También se pueden definir las matrices y vectores por medio de *funciones de librería* (las que se verán en la siguiente sección) y de *funciones programadas por el usuario* (que también se verán más adelante).

2.6. Operadores relacionales

El lenguaje de programación de MATLAB dispone de los siguientes operadores relacionales:

<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
==	igual que
~=	distinto que[8]

Obsérvese que, salvo el último de ellos, coinciden con los correspondientes operadores relacionales de C. Sin embargo, ésta es una coincidencia más bien formal. En MATLAB los operadores relacionales pueden aplicarse a vectores y matrices, y eso hace que tengan un significado especial.

Al igual que en C, si una comparación se cumple el resultado es 1 (***true***), mientras que si no se cumple es 0 (***false***). Recíprocamente, cualquier valor distinto de cero es considerado como ***true*** y el cero equivale a ***false***. La diferencia con C está en que cuando los operadores relacionales de MATLAB se aplican a dos matrices o vectores del mismo tamaño, *la comparación se realiza elemento a elemento*, y el resultado es otra matriz de unos y ceros del mismo tamaño, que recoge el resultado de cada comparación entre elementos. Considérese el siguiente ejemplo como ilustración de lo que se acaba de decir:

```
» A=[1 2;0 3]; B=[4 2;1 5];
» A==B
ans =
     0     1
     0     0
» A~=B
ans =
     1     0
     1     1
```

2.7. Operadores lógicos

Los operadores lógicos de MATLAB son los siguientes:

&	and
|	or
~	negación lógica

Obsérvese que estos operadores lógicos tienen distinta notación que los correspondientes operadores de C (&&, || y !). Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples. Más adelante se verán otros ejemplos y ciertas funciones de las que dispone MATLAB para facilitar la aplicación de estos operadores a vectores y matrices.

---

[8] El carácter (~) se obtiene en los PCs pulsando sucesivamente las teclas 1, 2 y 6 manteniendo ***Alt*** pulsada.

3. FUNCIONES DE LIBRERÍA

MATLAB tiene un gran número de funciones incorporadas. Algunas son ***funciones intrínsecas***, esto es, funciones incorporadas en el propio código ejecutable del programa. Estas funciones son particularmente rápidas y eficientes. Existen además funciones definidas en ficheros ****.m*** y ****.mex***[9] que vienen con el propio programa o que han sido aportadas por usuarios del mismo. Estas funciones extienden en gran manera las posibilidades del programa.

MATLAB 5.3 dispone también de ficheros ****.p***, que son los ficheros ****.m*** pre-compilados con la función ***pcode***. Se verán más adelante.

Recuérdese que para que MATLAB encuentre una determinada función de usuario el correspondiente *fichero-M* debe estar en el ***directorio actual*** o en uno de los directorios del ***search path***.

3.1. Características generales de las funciones de MATLAB

El concepto de función en MATLAB es semejante al de C y al de otros lenguajes de programación, aunque con algunas diferencias importantes. Al igual que en C, una función tiene ***nombre***, ***valor de retorno*** y ***argumentos***. Una función *se llama* utilizando su nombre en una expresión o utilizándolo como un comando más. Las funciones se pueden definir en ficheros de texto ****.m*** en la forma que se verá más adelante. Considérense los siguientes ejemplos de llamada a funciones:

```
» [maximo, posmax] = max(x);
» r = sqrt(x^2+y^2) + eps;
» a = cos(alfa) - sin(alfa);
```

donde se han utilizado algunas funciones matemáticas bien conocidas como el cálculo del valor máximo, el seno, el coseno y la raíz cuadrada. Los ***nombres*** de las funciones se han puesto en negrita. Los ***argumentos*** de cada función van a continuación del nombre entre paréntesis (y separados por comas si hay más de uno). Los ***valores de retorno*** son el resultado de la función y sustituyen a ésta en la expresión donde la función aparece.

Una diferencia importante con otros lenguajes es que en MATLAB las funciones pueden tener ***valores de retorno matriciales múltiples*** (ya se verá que pueden recogerse en variables *ad hoc* todos o sólo parte de estos valores de retorno), como en el primero de los ejemplos anteriores. En este caso se calcula el elemento de máximo valor en un vector, y se devuelven dos valores: el valor máximo y la posición que ocupa en el vector. Obsérvese que los 2 valores de retorno se recogen entre corchetes, separados por comas.

Una característica de MATLAB es que las funciones que no tienen argumentos no llevan paréntesis, por lo que a simple vista no siempre son fáciles de distinguir de las simples variables. En la segunda línea de los ejemplos anteriores, ***eps*** es una función sin argumentos, que devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. En lo sucesivo el nombre de la función irá seguido de paréntesis si interesa resaltar que la función espera que se le pase uno o más argumentos.

Los nombres de las funciones de MATLAB no son *palabras reservadas* del lenguaje. Es posible crear una variable llamada ***sin*** o ***cos***, que ocultan las funciones correspondientes. Para poder acceder a las funciones hay que eliminar (***clear***) las variables del mismo nombre que las ocultan.

MATLAB permite que una función tenga un número de argumentos y valores de retorno variable, determinado sólo en tiempo de ejecución. Más adelante se verá cómo se hace esto.

---

[9] Los ficheros ****.mex*** son ficheros de código ejecutable.

MATLAB tiene diversos tipos de funciones. A continuación se enumeran los tipos de funciones más importantes, clasificadas según su finalidad:

1.- Funciones matemáticas elementales.

2.- Funciones especiales.

3.- Funciones matriciales elementales.

4.- Funciones matriciales específicas.

5.- Funciones para la descomposición y/o factorización de matrices.

6.- Funciones para análisis estadístico de datos.

7.- Funciones para análisis de polinomios.

8.- Funciones para integración de ecuaciones diferenciales ordinarias.

9.- Resolución de ecuaciones no-lineales y optimización.

10.- Integración numérica.

11.- Funciones para procesamiento de señal.

A continuación se enumeran algunas características generales de las funciones de MATLAB:

– Los *argumentos actuales*[10] de estas funciones pueden ser expresiones y también llamadas a otra función.

– MATLAB nunca modifica las variables que se pasan como argumentos. Si el usuario las modifica dentro de la función, previamente se sacan copias de esas variables (se modifican las copias, no las variables originales).

– MATLAB admite valores de retorno matriciales múltiples. Por ejemplo, en el comando:

```
» [V, D] = eig(A)
```

la función ***eig()*** calcula los valores y vectores propios de la matriz cuadrada **A**. Los vectores propios se devuelven como columnas de la matriz **V**, mientras que los valores propios son los elementos de la matriz diagonal **D**. En los ejemplos siguientes:

```
» [xmax, imax] = max(x)
» xmax = max(x)
```

puede verse que la misma función ***max()*** puede ser llamada recogiendo dos valores de retorno (el máximo elemento de un vector y la posición que ocupa) o un sólo valor de retorno (el máximo elemento).

– Las operaciones de suma y/o resta de una matriz con un escalar consisten en sumar y/o restar el escalar a todos los elementos de la matriz.

– Recuérdese que tecleando ***help nombre_funcion*** se obtiene de inmediato información sobre la función de ese nombre. En el ***Help Desk*** aparecen enlaces a *"Matlab Functions by Subject"* y *"Matlab Functions by Index"*, en donde aparecen relaciones completas de las funciones disponibles en MATLAB.

[10] Los argumentos actuales son los que se utilizan en la llamada de la función

3.2. Equivalencia entre comandos y funciones

Existe una equivalencia entre las funciones y los comandos con argumentos de MATLAB. Así, un comando en la forma,

```
» comando arg1 arg2
```

es equivalente a una función con el mismo nombre que el comando a la que los argumentos se le pasan como cadenas de caracteres,

```
» comando('arg1', 'arg2')
```

Esta dualidad entre comandos y funciones es sobre todo útil en programación porque permite "*construir*" los argumentos con las operaciones propias de las cadenas de caracteres.

3.3. Funciones matemáticas elementales que operan de modo escalar

Estas funciones, que comprenden las funciones matemáticas trascendentales y otras funciones básicas, actúan sobre cada elemento de la matriz como si se tratase de un escalar. Se aplican de la misma forma a escalares, vectores y matrices. Algunas de las funciones de este grupo son las siguientes:

sin(x)	seno
cos(x)	coseno
tan(x)	tangente
asin(x)	arco seno
acos(x)	arco coseno
atan(x)	arco tangente (devuelve un ángulo entre $-\pi/2$ y $+\pi/2$)
atan2(x)	arco tangente (devuelve un ángulo entre $-\pi$ y $+\pi$); se le pasan 2 argumentos, proporcionales al seno y al coseno
sinh(x)	seno hiperbólico
cosh(x)	coseno hiperbólico
tanh(x)	tangente hiperbólica
asinh(x)	arco seno hiperbólico
acosh(x)	arco coseno hiperbólico
atanh(x)	arco tangente hiperbólica
log(x)	logaritmo natural
log10(x)	logaritmo decimal
exp(x)	función exponencial
sqrt(x)	raíz cuadrada
sign(x)	devuelve -1 si <0, 0 si =0 y 1 si >0. Aplicada a un número complejo, devuelve un vector unitario en la misma dirección
rem(x,y)	resto de la división (2 argumentos que no tienen que ser enteros)
mod(x,y)	similar a ***rem*** (Ver diferencias con el ***Help***)
round(x)	redondeo hacia el entero más próximo
fix(x)	redondea hacia el entero más próximo a 0
floor(x)	valor entero más próximo hacia $-\infty$
ceil(x)	valor entero más próximo hacia $+\infty$
gcd(x)	máximo común divisor
lcm(x)	mínimo común múltiplo
real(x)	partes reales
imag(x)	partes imaginarias
abs(x)	valores absolutos
angle(x)	ángulos de fase

3.4. Funciones que actúan sobre vectores

Las siguientes funciones actúan sobre vectores (no sobre matrices ni sobre escalares)

[xm,im]=max(x)	máximo elemento de un vector. Devuelve el valor máximo **xm** y la posición que ocupa **im**
min(x)	mínimo elemento de un vector. Devuelve el valor mínimo y la posición que ocupa
sum(x)	suma de los elementos de un vector
cumsum(x)	devuelve el vector suma acumulativa de los elementos de un vector (cada elemento del resultado es una suma de elementos del original)
mean(x)	valor medio de los elementos de un vector
std(x)	desviación típica
prod(x)	producto de los elementos de un vector
cumprod(x)	devuelve el vector producto acumulativo de los elementos de un vector
[y,i]=sort(x)	ordenación de menor a mayor de los elementos de un vector **x**. Devuelve el vector ordenado **y**, y un vector **i** con las posiciones iniciales en **x** de los elementos en el vector ordenado **y**.

En realidad estas funciones *se pueden aplicar también a matrices*, pero en ese caso se aplican por separado a cada columna de la matriz, dando como valor de retorno un vector resultado de aplicar la función a cada columna de la matriz considerada como vector. Si estas funciones se quieren aplicar a las filas de la matriz basta aplicar dichas funciones a la matriz traspuesta.

3.5. Funciones que actúan sobre matrices

Las siguientes funciones exigen que el/los argumento/s sean matrices. En este grupo aparecen algunas de las funciones más útiles y potentes de MATLAB. Se clasificarán en varios subgrupos:

3.5.1. FUNCIONES MATRICIALES ELEMENTALES:

B = A'	calcula la traspuesta (conjugada) de la matriz **A**
B = A.'	calcula la traspuesta (sin conjugar) de la matriz **A**
v = poly(A)	devuelve un vector **v** con los coeficientes del polinomio característico de la matriz cuadrada **A**
t = trace(A)	devuelve la traza **t** (suma de los elementos de la diagonal) de una matriz cuadrada **A**
[m,n] = size(A)	devuelve el número de filas **m** y de columnas **n** de una matriz rectangular **A**
n = size(A)	devuelve el tamaño de una matriz cuadrada **A**
nf = size(A,1)	devuelve el número de filas de **A**
nc = size(A,2)	devuelve el número de columnas de **A**

3.5.2. FUNCIONES MATRICIALES ESPECIALES

Las funciones ***exp()***, ***sqrt()*** y ***log()*** se aplican elemento a elemento a las matrices y/o vectores que se les pasan como argumentos. Existen otras funciones similares que tienen también sentido cuando se aplican a una matriz como una única entidad. Estas funciones son las siguientes (se distinguen porque llevan una "m" adicional en el nombre):

expm(A)	si A=XDX', expm(A) = X*diag(exp(diag(D)))*X'
sqrtm(A)	devuelve una matriz que multiplicada por sí misma da la matriz **A**
logm()	es la función recíproca de expm(A)

Aunque no pertenece a esta familia de funciones, se puede considerar que el ***operador potencia*** (^) está emparentado con ellas. Así, es posible decir que:

A^n	está definida si **A** es cuadrada y **n** un número real. Si **n** es entero, el resultado se calcula por multiplicaciones sucesivas. Si **n** es real, el resultado se calcula como: A^n=X*D.^n*X' siendo [X,D]=eig(A)

3.5.3. FUNCIONES DE FACTORIZACIÓN Y/O DESCOMPOSICIÓN MATRICIAL

A su vez este grupo de funciones se puede subdividir en 4 subgrupos:

– Funciones basadas en la factorización triangular (eliminación de Gauss):

[L,U] = lu(A)	descomposición de Crout (A = LU) de una matriz. La matriz **L** es una permutación de una matriz triangular inferior (dicha permutación es consecuencia del pivotamiento por columnas utilizado en la factorización)
B = inv(A)	calcula la inversa de **A**. Equivale a B=inv(U)*inv(L)
d = det(A)	devuelve el determinante **d** de la matriz cuadrada **A**. Equivale a d=det(L)*det(U)
E = rref(A)	reducción a forma de escalón (mediante la eliminación de Gauss con pivotamiento por columnas) de una matriz rectangular **A**
[E,xc] = rref(A)	reducción a forma de escalón con un vector **xc** que da información sobre una posible base del espacio de columnas de **A**
U = chol(A)	descomposición de Cholesky de matriz simétrica y positivo-definida. Sólo se utiliza la diagonal y la parte triangular superior de **A**. El resultado es una matriz triangular superior tal que A = U'*U
c = rcond(A)	devuelve una estimación del recíproco de la condición numérica de la matriz **A** basada en la norma sub-1. Si el resultado es próximo a 1 la matriz **A** está bien condicionada; si es próximo a 0 no lo está.

– Funciones basadas en el cálculo de valores y vectores propios:

[X,D] = eig(A)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de una matriz cuadrada **A**. Con frecuencia el resultado es complejo (si **A** no es simétrica)
[X,D] = eig(A,B)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de dos matrices cuadradas **A** y **B** (**Ax** = λ**Bx**).

– Funciones basadas en la descomposición QR:

[Q,R] = qr()	descomposición QR de una matriz rectangular. Se utiliza para sistemas con más ecuaciones que incógnitas.
B = null(A)	devuelve una base ortonormal del subespacio nulo (kernel, o conjunto de vectores **x** tales que **Ax** = 0) de la matriz rectangular **A**
Q = orth(A)	las columnas de **Q** son una base ortonormal del espacio de columnas de **A**. El número de columnas de **Q** es el rango de **A**

– Funciones basadas en la descomposición de valor singular

[U,D,V] = svd(A)	descomposición de valor singular de una matriz rectangular (A=U*D*V'). **U** y **V** son matrices ortonormales. **D** es diagonal y contiene los valores singulares
B = pinv(A)	calcula la pseudo-inversa de una matriz rectangular **A**
r = rank(A)	calcula el rango **r** de una matriz rectangular **A**

nor = norm(A)	calcula la norma sub-2 de una matriz (el mayor valor singular)
nor = norm(A,2)	lo mismo que la anterior
c = cond(A)	condición numérica sub-2 de la matriz **A**. Es el cociente entre el máximo y el mínimo valor singular. La condición numérica da una idea de los errores que se obtienen al resolver un sistema de ecuaciones lineales con dicha matriz: su logaritmo indica el número de cifras significativas que se pierden.

– Cálculo del rango, normas y condición numérica:

Existen varias formas de realizar estos cálculos, con distintos niveles de esfuerzo de cálculo y de precisión en el resultado.

El rango se calcula implícitamente (sin que el usuario lo pida) al ejecutar las funciones ***rref(A)***, ***orth(A)***, ***null(A)*** y ***pinv(A)***. Con ***rref(A)*** el rango se calcula como el número de filas diferentes de cero; con ***orth(A)*** y ***null(A)*** –basadas ambas en la descomposición QR– el rango es el número de columnas del resultado (o **n** menos el número de columnas del resultado). Con ***pinv(A)*** se utiliza la descomposición de valor singular, que es el método más fiable y más caro en tiempo de *cpu*. La función ***rank(A)*** está basada en ***pinv(A)***.

Normas de matrices:

norm(A)	norma sub-2, es decir, máximo valor singular de **A**, ***max(svd(A))***.
normest(A)	calcula una estimación o aproximación de la norma sub-2. Útil para matrices grandes en las que ***norm(A)*** necesita demasiado tiempo
norm(A,2)	lo mismo que ***norm(A)***
norm(A,1)	norma sub-1 de **A**, máxima suma de valores absolutos por columnas, es decir: ***max(sum(abs((A))))***
norm(A,inf)	norma sub-∞ de **A**, máxima suma de valores absolutos por filas, es decir: ***max(sum(abs((A'))))***

Normas de vectores:

norm(x,p)	norma sub-p, es decir ***sum(abs(x)^p)^(1/p)***.
norm(x)	norma euclídea; equivale al módulo o ***norm(x,2)***.
norm(x,inf)	norma sub-∞, es decir ***max(abs(x))***.
norm(x,1)	norma sub-1, es decir ***sum(abs(x))***.

3.6. Más sobre operadores relacionales con vectores y matrices

Cuando alguno de los operadores relacionales vistos previamente (<, >, <=, >=, == y ~=) actúa entre dos matrices (vectores) del mismo tamaño, el resultado es otra matriz (vector) de ese mismo tamaño conteniendo unos y ceros, según los resultados de cada comparación entre elementos hayan sido ***true*** o ***false***, respectivamente.

Por ejemplo, supóngase que se define una matriz *magic* **A** de tamaño 3x3 y a continuación se forma una matriz binaria **M** basada en la condición de que los elementos de **A** sean mayores que 4 (MATLAB convierte este cuatro en una matriz de cuatros de modo automático). Obsérvese con atención el resultado:

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
```

```
» M=A>4
M =
     1     0     1
     0     1     1
     0     1     0
```

De ordinario, las matrices "binarias" que se obtienen de la aplicación de los operadores relacionales no se almacenan en memoria ni se asignan a variables, sino que se procesan sobre la marcha. MATLAB dispone de varias funciones para ello. Recuérdese que cualquier valor distinto de cero equivale a ***true***, mientras que un valor cero equivale a ***false***. Algunas de estas funciones son:

any(x)	función vectorial; chequea si *alguno* de los elementos del vector **x** cumple una determinada condición (en este caso ser distinto de cero). Devuelve un uno ó un cero
any(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
all(x)	función vectorial; chequea si *todos* los elementos del vector **x** cumplen una condición. Devuelve un uno ó un cero
all(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
find(x)	busca índices correspondientes a elementos de vectores que cumplen una determinada condición. El resultado es un vector con los índices de los elementos que cumplen la condición
find(A)	cuando esta función se aplica a una matriz la considera como un vector con una columna detrás de otra, de la 1ª a la última.

A continuación se verán algunos ejemplos de utilización de estas funciones.

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
» m=find(A>4)
m =
     1
     5
     6
     7
     8
```

Ahora se van a sustituir los elementos que cumplen la condición anterior por valores de 10. Obsérvese cómo se hace y qué resultado se obtiene:

```
» A(m)=10*ones(size(m))
A =
    10     1    10
     3    10    10
     4    10     2
```

donde ha sido necesario convertir el 10 en un vector del mismo tamaño que **m**. Para chequear si hay algún elemento de un determinado valor –por ejemplo 3– puede hacerse lo siguiente:

```
» any(A==3)
ans =
     1     0     0
» any(ans)
ans =
     1
```

mientras que para comprobar que todos los elementos de **A** son mayores que cero:

```
» all(all(A))
ans =
     1
```

En este caso no ha hecho falta utilizar el operador relacional porque cualquier elemento distinto de cero equivale a ***true***.

La función ***isequal(A, B)*** devuelve *uno* si las matrices son idénticas y *cero* si no lo son.

3.7. Otras funciones que actúan sobre vectores y matrices

Las siguientes funciones pueden actuar sobre vectores y matrices, y sirven para chequear ciertas condiciones:

exist(var)	comprueba si la variable **var** existe
isnan()	chequea si hay valores ***NaN***, devolviendo una matriz de unos y ceros
isinf()	chequea si hay valores ***Inf***, devolviendo una matriz de unos y ceros
isfinite()	chequea si los valores son finitos
isempty()	chequea si un vector o matriz está vacío
ischar()	chequea si una variable es una cadena de caracteres (*string*)
isglobal()	chequea si una variable es global
issparse()	chequea si una matriz es dispersa (*sparse*, es decir, con un gran número de elementos cero)

A continuación se presentan algunos ejemplos de uso de estas funciones en combinación con otras vistas previamente. Se define un vector **x** con un ***NaN***, que se elimina en la forma:

```
» x=[1 2 3 4 0/0 6]
Warning: Divide by zero
x =
     1     2     3     4   NaN     6
» i=find(isnan(x))
i =
     5
» x=x(find(~isnan(x)))
x =
     1     2     3     4     6
```

Otras posibles formas de eliminarlo serían las siguientes:

```
» x=x(~isnan(x))
» x(isnan(x))=[]
```

La siguiente sentencia elimina las filas de una matriz que contienen algún ***NaN***:

```
» A(any(isnan(A)'), :)=[]
```

3.8. Determinación de la fecha y la hora

MATLAB dispone de funciones que dan información sobre la ***fecha*** y la ***hora*** actual (la del reloj del ordenador). Las funciones más importantes relacionadas con la fecha y la hora son las siguientes.

clock	devuelve un vector fila de seis elementos que representan el ***año***, el ***mes***, el ***día***, la ***hora***, los ***minutos*** y los ***segundos***, según el reloj interno del computador. Los cinco primeros son valores enteros, pero la cifra correspondiente a los segundos contiene información hasta las milésimas de segundo.

now	devuelve un número (***serial date number***) que contiene toda la información de la fecha y hora actual. Se utiliza como argumento de otras funciones.
date	devuelve la fecha actual como cadena de caracteres (por ejemplo: *24-Aug-1999*).
datestr(t)	convierte el *serial date number **t*** en cadena de caracteres con el *día*, *mes*, *año*, *hora*, *minutos* y *segundos*. Ver en los manuales on-line los formatos de cadena admitidos.
datenum()	convierte una cadena ('mes-día-año') o un conjunto de seis números (año, mes, día, horas, minutos, segundos) en *serial date number*.
datevec()	convierte *serial date numbers* o cadenas de caracteres en el vector de seis elementos que representa la fecha y la hora.
calendar()	devuelve una matriz 6x7 con el calendario del mes actual, o del mes y año que se especifique como argumento.
weekday(t)	devuelve el día de la semana para un *serial date number **t***.

3.9. Funciones para cálculos con polinomios

Para MATLAB un polinomio se puede definir mediante un vector de coeficientes. Por ejemplo, el polinomio:

$$x^4 - 8x^2 + 6x - 10 = 0$$

se puede representar mediante el vector [1, 0, -8, 6, -10]. MATLAB puede realizar diversas operaciones sobre él, como por ejemplo evaluarlo para un determinado valor de **x** (función ***polyval()***) y calcular las raíces (función ***roots()***):

```
» pol=[1 0 -8 6 -10]
pol =
     1     0    -8     6   -10
» roots(pol)
ans =
  -3.2800
   2.6748
   0.3026 + 1.0238i
   0.3026 - 1.0238i
» polyval(pol,1)
ans =
   -11
```

Para calcular producto de polinomios MATLAB utiliza una función llamada ***conv()*** (de *producto de convolución*). En el siguiente ejemplo se va a ver cómo se multiplica un polinomio de segundo grado por otro de tercer grado:

```
» pol1=[1 -2 4]
pol1 =
     1    -2     4
» pol2=[1 0 3 -4]
pol2 =
     1     0     3    -4
» pol3=conv(pol1,pol2)
pol3 =
     1    -2     7   -10    20   -16
```

Para dividir polinomios existe otra función llamada ***deconv()***. Las funciones orientadas al cálculo con polinomios son las siguientes:

poly(A)	polinomio característico de la matriz **A**
roots(pol)	raíces del polinomio **pol**
polyval(pol,x)	evaluación del polinomio **pol** para el valor de **x**. Si **x** es un vector, **pol** se evalúa para cada elemento de **x**
polyvalm(pol,A)	evaluación del polinomio **pol** de la matriz **A**
conv(p1,p2)	producto de convolución de dos polinomios **p1** y **p2**
[c,r]=deconv(p,q)	división del polinomio **p** por el polinomio **q**. En **c** se devuelve el cociente y en **r** el resto de la división
residue(p1,p2)	descompone el cociente entre **p1** y **p2** en suma de fracciones simples (ver ***»help residue***)
polyder(pol)	calcula la derivada de un polinomio
polyder(p1,p2)	calcula la derivada de producto de polinomios
polyfit(x,y,n)	calcula los coeficientes de un polinomio **p(x)** de grado **n** que se ajusta a los datos **p(x(i)) ~= y(i)**, en el sentido de mínimo error cuadrático medio.

4. OTROS TIPOS DE DATOS DE MATLAB

En los Capítulos precedentes se ha visto la "especialidad" de MATLAB: trabajar con vectores y matrices. En este Capítulo se va a ver que MATLAB puede también trabajar con otros tipos de datos:

1. Conjuntos o cadenas de caracteres, fundamentales en cualquier lenguaje de programación.
2. Hipermatrices, o matrices de más de dos dimensiones.
3. Estructuras, o agrupaciones bajo un mismo nombre de datos de naturaleza diferente.
4. Vectores o matrices de celdas (cell arrays), que son vectores o matrices cuyos elementos pueden ser cualquier otro tipo de dato.
5. Matrices dispersas o matrices dispersas, que son matrices que pueden ser de muy gran tamaño con la mayor parte de sus elementos cero.

4.1. Cadenas de caracteres

MATLAB trabaja también con ***cadenas de caracteres***, con ciertas semejanzas y también diferencias respecto a C/C++ y Java. A continuación se explica lo más importante del manejo de cadenas de caracteres en MATLAB. Las funciones para cadenas de caracteres están en el sub-directorio ***toolbox\matlab\strfun*** del directorio en que esté instalado MATLAB.

Los caracteres de una cadena se almacenan en un vector, con un carácter por elemento. Cada carácter ocupa dos bytes. Las cadenas de caracteres van entre ***apóstrofos*** o ***comillas simples***, como por ejemplo: 'cadena'. Si la cadena debe contener comillas, éstas se representan por un doble carácter comilla, de modo que se pueden distinguir fácilmente del principio y final de la cadena. Por ejemplo, para escribir la cadena **ni 'idea'** se escribiría **'ni''idea'''**.

Una ***matriz de caracteres*** es una matriz cuyos elementos son caracteres, o bien una matriz cuyas filas son cadenas de caracteres. Todas las filas de una ***matriz de caracteres*** deben tener el ***mismo número de elementos***. Si es preciso, las cadenas (filas) más cortas se completan con blancos.

A continuación se pueden ver algunos ejemplos y practicar con ellos:

```
» c='cadena'
c =
cadena
» size(c)            % dimensiones del array
ans =
     1     6
» double(c)          % convierte en números ASCII cada carácter
ans =
    99    97   100   101   110    97
» char(abs(c))       % convierte números ASCII en caracteres
ans =
cadena
» cc=char('más','madera')   % convierte dos cadenas en una matriz
cc =
más
madera
» size(cc)           % se han añadido tres espacios a 'más'
ans =
     2     6
```

Las funciones más importantes para manejo de cadenas de caracteres son las siguientes:

double(c)	convierte en números ASCII cada carácter
char(v)	convierte un vector de números **v** en una cadena de caracteres
char(c1,c2)	crea una matriz de caracteres, completando con blancos las cadenas más cortas
deblank(c)	elimina los blancos al final de una cadena de caracteres
disp(c)	imprime el texto contenido en la variable **c**
ischar(c)	detecta si una variable es una cadena de caracteres
isletter()	detecta si un carácter es una letra del alfabeto. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros
isspace()	detecta si un carácter es un espacio en blanco. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros
strcmp(c1,c2)	comparación de cadenas. Si las cadenas son iguales devuelve un uno, y si no lo son, devuelve un cero (funciona de modo diferente que la correspondiente función de C)
strcmpi(c1,c2)	igual que ***strcmp(c1,c2)***, pero ignorando la diferencia entre mayúsculas y minúsculas
strncmp(c1,c2,n)	compara los **n** primeros caracteres de dos cadenas
c1==c2	compara dos cadenas carácter a carácter. Devuelve un vector o matriz de unos y ceros
s=[s,' y más']	concatena cadenas, añadiendo la segunda a continuación de la primera
findstr(c1,c2)	devuelve un vector con las posiciones iniciales de todas las veces en que la cadena más corta aparece en la más larga
strmatch(cc,c)	devuelve los índices de todos los elementos de la matriz de caracteres (o vector de celdas) **cc**, que empiezan por la cadena **c**
strrep(c1,c2,c3)	sustituye la cadena **c2** por **c3**, cada vez que **c2** es encontrada en **c1**
[p,r]=strtok(t)	separa las palabras de una cadena de caracteres **t**. Devuelve la primera palabra **p** y el resto de la cadena **r**
int2str(v)	convierte un número entero en cadena de caracteres
num2str(x,n)	convierte un número real **x** en su expresión por medio de una cadena de caracteres, con cuatro cifras decimales por defecto (pueden especificarse más cifras, con un argumento opcional **n**)
str2double(str)	convierte una cadena de caracteres representando un número real en el número real correspondiente
vc=cellstr(cc)	convierte una matriz de caracteres **cc** en un vector de celdas **vc**, eliminando los blancos adicionales al final de cada cadena. La función ***char()*** realiza las conversiones opuestas
sprintf	convierte valores numéricos en cadenas de caracteres, de acuerdo con las reglas y formatos de conversión del lenguaje C. Esta es la función más general para este tipo de conversión y se verá con mas detalle en la Sección 5.5.2.

Con las funciones anteriores se dispone en MATLAB de una amplia gama de posibilidades para trabajar con cadenas de caracteres.

A continuación se pueden ver algunos ejemplos:

```
» num2str(pi)   % el resultado es una cadena de caracteres, no un número
ans =
3.142
» num2str(pi,8)
ans =
3.1415927
```

Es habitual convertir los valores numéricos en cadenas de caracteres para poder imprimirlos como títulos en los dibujos o gráficos. Véase el siguiente ejemplo:

```
» fahr=70; grd=(fahr-32)/1.8;
» title(['Temperatura ambiente: ',num2str(grd),' grados centígrados'])
```

4.2. Hipermatrices (arrays de más de dos dimensiones)

MATLAB permite trabajar con ***hipermatrices***, es decir con matrices de más de dos dimensiones (Figura 11). Una posible aplicación es almacenar con un único nombre distintas matrices del mismo tamaño (resulta una hipermatriz de 3 dimensiones). Los elementos de una hipermatriz pueden ser números, caracteres, estructuras, y vectores o matrices de celdas.

El tercer subíndice representa la tercera dimensión: la "profundidad" de la hipermatriz.

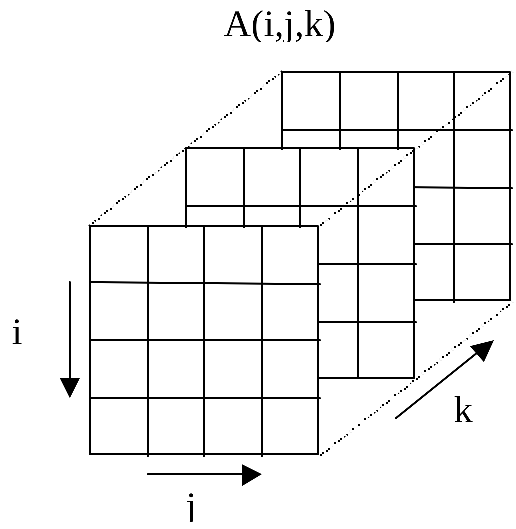


Figura 11. Hipermatriz de tres dimensiones.

4.2.1. DEFINICIÓN DE HIPERMATRICES

Las funciones para trabajar con estas hipermatrices están en el sub-directorio ***toolbox\matlab\datatypes***. Las funciones que operan con matrices de más de dos dimensiones son análogas a las funciones vistas previamente, aunque con algunas diferencias. Por ejemplo, las siguientes sentencias generan, en dos pasos, una matriz de 2x3x2:

```
» AA(:,:,1)=[1 2 3; 4 5 6]   % matriz inicial
AA =
     1     2     3
     4     5     6
» AA(:,:,2)=[2 3 4; 5 6 7]   % se añade una segunda matriz
AA(:,:,1) =
     1     2     3
     4     5     6
AA(:,:,2) =
     2     3     4
     5     6     7
```

4.2.2. FUNCIONES QUE TRABAJAN CON HIPERMATRICES

Algunas funciones de MATLAB para generar matrices admiten más de dos subíndices y pueden ser utilizadas para generar hipermatrices. Entre ellas están ***rand()***, ***randn()***, ***zeros()*** y ***ones()***. Por ejemplo, véase la siguiente sentencia y su resultado:

```
» BB=randn(2,3,2)
BB(:,:,1) =
   -0.4326    0.1253   -1.1465
   -1.6656    0.2877    1.1909
BB(:,:,2) =
    1.1892    0.3273   -0.1867
   -0.0376    0.1746    0.7258
```

La función ***cat()*** permite concatenar matrices según las distintas "dimensiones", como puede verse en el siguiente ejemplo:

```
» A=zeros(2,3); B=ones(2,3);
```

```
» cat(1,A,B)
ans =
     0     0     0
     0     0     0
     1     1     1
     1     1     1
» cat(2,A,B)
ans =
     0     0     0     1     1     1
     0     0     0     1     1     1
» cat(3,A,B)
ans(:,:,1) =
     0     0     0
     0     0     0
ans(:,:,2) =
     1     1     1
     1     1     1
```

Las siguientes funciones de MATLAB se pueden emplear también con hipermatrices:

size()	devuelve tres o más valores (el nº de elementos en cada dimensión)
ndims()	devuelve el número de dimensiones
squeeze()	elimina las dimensiones que son igual a uno
reshape()	distribuye el mismo número de elementos en una matriz con distinta forma o con distintas dimensiones
permute(A,v)	permuta las dimensiones de **A** según los índices del vector **v**
ipermute(A,v)	realiza la permutación inversa

Respecto al resto de las funciones de MATLAB, se pueden establecer las siguientes reglas para su aplicación a hipermatrices:

1. Todas las funciones de MATLAB que operan sobre escalares (***sin()***, ***cos()***, etc.) se aplican sobre hipermatrices elemento a elemento (igual que sobre vectores y matrices). Las operaciones con escalares también se aplican de la misma manera.
2. Las funciones que operan sobre vectores (***sum()***, ***max()***, etc.) se aplican a matrices e hipermatrices según la primera dimensión, resultando un array de una dimensión inferior.
3. Las funciones matriciales propias del Álgebra Lineal (***det()***, ***inv()***, etc.) no se pueden aplicar a hipermatrices. Para poderlas aplicar hay que extraer primero las matrices correspondientes (por ejemplo, con el operador dos puntos (:)).

4.3. Estructuras

Una estructura (***struct***) es una agrupación de datos de tipo diferente bajo un mismo nombre. Estos datos se llaman ***miembros*** (*members*) o ***campos*** (*fields*). Una estructura es un nuevo tipo de dato, del que luego se pueden crear muchas variables (***objetos*** o *instances*). Por ejemplo, la estructura ***alumno*** puede contener los campos ***nombre*** (una cadena de caracteres) y ***carnet*** (un número).

4.3.1. Creación de estructuras

En MATLAB la estructura ***alumno*** se crea creando un objeto de dicha estructura. A diferencia de otros lenguajes de programación, no hace falta definir previamente el modelo o patrón de la estructura. Una posible forma de hacerlo es crear uno a uno los distintos campos, como en el ejemplo siguiente:

```
» alu.nombre='Miguel'
alu =
    nombre: 'Miguel'
» alu.carnet=75482
alu =
    nombre: 'Miguel'
    carnet: 75482
» alu
alu =
    nombre: 'Miguel'
    carnet: 75482
```

Se accede a los miembros o campos de una estructura por medio del ***operador punto*** (.), que une el nombre de la estructura y el nombre del campo (por ejemplo: ***alu.nombre***).

También puede crearse la estructura por medio de la función ***struct()***, como por ejemplo,

```
» al = struct('nombre', 'Ignacio', 'carnet', 76589)
al =
    nombre: 'Ignacio'
    carnet: 76589
```

Los ***nombres de los campos*** se pasan a la función ***struct()*** entre apóstrofos ('), seguidos del valor que se les quiere dar. Este valor puede ser la cadena vacía ('') o la matriz vacía ([]).

Pueden crearse vectores y matrices (e hipermatrices) de estructuras. Por ejemplo, la sentencia,

```
» alum(10) = struct('nombre', 'Ignacio', 'carnet', 76589)
```

crea un vector de 10 elementos cada uno de los cuales es una estructura tipo ***alumno***. Sólo el elemento 10 del vector es inicializado con los argumentos de la f***unción struct()***; el resto de los campos se inicializan con una cadena vacía o una matriz vacía[11]. Para dar valor a los campos de los elementos restantes se puede utilizar un bucle ***for*** con sentencias del tipo:

```
» alum(i).nombre='Noelia', alum(i).carnet=77524;
```

MATLAB permite añadir un nuevo campo a una estructura en cualquier momento. La siguiente sentencia añade el campo ***edad*** a todos los elementos del vector ***alum***, aunque sólo se da valor al campo del elemento 5,

```
» alum(5).edad=18;
```

Para ver el campo ***edad*** en los 10 elementos del vector puede teclearse el comando:

```
» alum.edad
```

4.3.2. FUNCIONES PARA OPERAR CON ESTRUCTURAS

Las estructuras de MATLAB disponen de funciones que facilitan su uso. Algunas de estas funciones son las siguientes:

fieldnames()	devuelve un vector de celdas con cadenas de caracteres que recogen los nombres de los campos de una estructura
isfield(ST,s)	permite saber si la cadena **s** es un campo de una estructura **ST**
isstruct(ST)	permite saber si **ST** es o no una estructura
rmfield(ST,s)	elimina el campo **s** de la estructura **ST**
getfield(ST,s)	devuelve el valor del campo especificado. Si la estructura es un array hay que pasarle los índices como *cell array* (entre llaves {}) como segundo argumento

[11] Esta forma de crear arrays de estructuras da error si la estructura ha sido previamente declarada ***global***.

setfield(ST,s,v) da el valor **v** al campo **s** de la estructura **ST**. Si la estructura es un array, hay que pasarle los índices como *cell array (*entre llaves {}) como segundo argumento

MATLAB permite definir ***estructuras anidadas***, es decir una estructura con campos que sean otras estructuras. Para acceder a los campos de la estructura más interna se utiliza dos veces el operador punto (.), como puede verse en el siguiente ejemplo, en el que la estructura ***clase*** contiene un campo que es un vector ***alum*** de alumnos,

```
» clase=struct('curso','primero','grupo','A', ...
               'alum', struct('nombre','Juan', 'edad', 19))
clase =
    curso: 'primero'
    grupo: 'A'
     alum: [1x1 struct]
» clase.alum(2).nombre='María';
» clase.alum(2).edad=17;
» clase.alum(2)
ans =
    nombre: 'María'
      edad: 17
» clase.alum(1)
ans =
    nombre: 'Juan'
      edad: 19
```

Las estructuras se generalizan con las ***clases*** y los ***objetos***, que no se verán en este manual.

4.4. Vectores o matrices de celdas (*Cell Arrays*)

Un vector (matriz o hipermatriz) de celdas es un vector (matriz o hipermatriz) cuyos elementos son cada uno de ellos una variable de tipo cualquiera. En un array ordinario todos sus elementos son números o cadenas de caracteres. Sin embargo, en un ***array de celdas***, el primer elemento puede ser un número; el segundo una matriz; el tercero una cadena de caracteres; el cuarto una estructura, etc.

4.4.1. CREACIÓN DE VECTORES Y MATRICES DE CELDAS

Obsérvese por ejemplo cómo se crea, utilizando ***llaves*** {}, el siguiente vector de celdas,

```
» vc(1)={[1 2 3]}
vc =
    [1x3 double]
» vc(2)={'mi nombre'}
vc =
    [1x3 double]    'mi nombre'
» vc(3)={rand(3,3)}
vc =
    [1x3 double]    'mi nombre'    [3x3 double]
```

Es importante que el nombre del vector de celdas ***vc*** no haya sido utilizado previamente para otra variable (si así fuera, se obtendría un error). Si es preciso se utiliza el comando ***clear***.

Obsérvese que para crear un vector de celdas los valores asignados a cada elemento se han definido entre ***llaves*** {...}.

Otra nomenclatura alternativa y similar, que también utiliza llaves, es la que se muestra a continuación:

```
» vb{1}=[1 2 3]
vb =
    [1x3 double]
» vb{2}='mi nombre'
vb =
    [1x3 double]    'mi nombre'
» vb{3}=rand(3,3)
vb =
    [1x3 double]    'mi nombre'    [3x3 double]
```

y también es posible crear el vector de celdas en una sola operación en la forma,

```
vcc = {[1 2 3], 'mi nombre', rand(3,3)}
vcc =
    [1x3 double]    'mi nombre'    [3x3 double]
```

4.4.2. FUNCIONES PARA TRABAJAR CON VECTORES Y MATRICES DE CELDAS

MATLAB dispone de las siguientes funciones para trabajar con *cell arrays*:

cell(m,n)	crea un *cell array* vacío de **m** filas y **n** columnas
celldisp(ca)	muestra el contenido de todas las celdas de **ca**
cellplot(ca)	muestra una representación gráfica de las distintas celdas
iscell(ca)	indica si **ca** es un vector de celdas
num2cell()	convierte un array numérico en un *cell array*
cell2struct()	convierte un *cell array* en una estructura (ver Sección 4.4.3)
struct2cell()	convierte una estructura en un *cell array* (ver Sección 4.4.3)

4.4.3. CONVERSIÓN ENTRE ESTRUCTURAS Y VECTORES DE CELDAS

El siguiente ejemplo convierte el *cell array* ***vcc*** creado previamente en una estructura ***ST*** cuyos ***campos*** se pasan como argumentos a la función ***cell2struct()***. El tercer argumento (un 2) indica que es la segunda dimensión del *cell array* (las columnas) la que va a dar origen a los campos de la estructura. Con posterioridad la estructura ***ST*** se convierte en un nuevo *cell array* llamado ***vbb***,

```
» ST=cell2struct(vb,{'vector','cadena','matriz'},2)
ST =
    vector: [1 2 3]
    cadena: 'mi nombre'
    matriz: [3x3 double]
» vbb = struct2cell(ST)'    % hay que transponer para obtener una fila
vbb =
    [1x3 double]    'mi nombre'    [3x3 double]
```

La gran ventaja de las estructuras y los arrays de celdas es que proporcionan una gran flexibilidad para el almacenamiento de los más diversos tipos de información. El inconveniente es que se pierde parte de la eficiencia que MATLAB tiene trabajando con vectores y matrices.

4.5. Matrices dispersas (sparse)

Las matrices dispersas o sparse son matrices de un gran tamaño con la mayor parte de sus elementos cero. Operar sobre este tipo de matrices con los métodos convencionales lleva a obtener tiempos de cálculo prohibitivos. Por esta razón se han desarrollado técnicas especiales para este tipo de matrices. En ingeniería es muy frecuente encontrar aplicaciones en las que aparecen matrices sparse. MATLAB dispone de numerosas funciones para trabajar con estas matrices.

Las matrices dispersas se almacenan de una forma especial: solamente se guardan en memoria los elementos distintos de cero, junto con la posición que ocupan en la matriz. MATLAB usa 3 arrays para matrices reales sparse con ***nnz*** elementos distintos de cero:

1. Un array con todos los elementos distintos de cero (*nnz* elementos)
2. Un array con los índices de fila de los elementos distintos de cero (*nnz* elementos)
3. Un array con punteros a la posición del primer elemento de cada columna (*n* elementos)

En total se requiere una memoria de (*nnz**8+(*nnz*+*n*)*4) bytes. La Figura 12 muestra un ejemplo de matriz dispersa que viene con MATLAB (se puede cargar con ***load west0479***). Esta matriz tiene 479 filas y columnas. De los 229441 elementos sólo 1887 son distintos de cero. Se comprende que se pueden conseguir grandes ahorros de memoria y de tiempo de cálculo almacenando y operando sólo con los elementos distintos de cero.

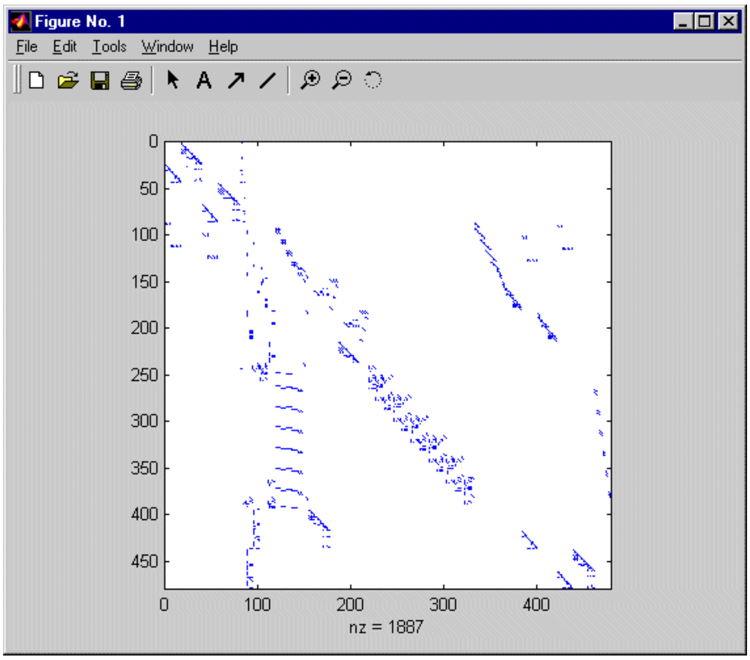


Figura 12. Ejemplo de matriz dispersa (west0479).

A continuación se va a mostrar con un ejemplo más pequeño cómo guarda MATLAB estas matrices. Primero se creará una matriz 5×5 llena y luego se convertirá en dispersa.

```
» A=[1, 0, 0, -1, 0;  0, 2, 0, 0, 1;  0, 0, 1, 1, 0; 0, 2, 0, 1, 0;  -3, 0, 0, 0, 2]
A =
     1     0     0    -1     0
     0     2     0     0     1
     0     0     1     1     0
     0     2     0     1     0
    -3     0     0     0     2
» S=sparse(A)
S =
   (1,1)        1
   (5,1)       -3
   (2,2)        2
   (4,2)        2
   (3,3)        1
   (1,4)       -1
   (3,4)        1
   (4,4)        1
   (2,5)        1
   (5,5)        2
```

Puede observarse cómo MATLAB muestra las matrices dispersas: primero los dos índices, de filas y de columnas, entre paréntesis y después el valor del elemento. Los elementos se almacenan por columnas y por eso se guarda la posición en que empieza cada columna.

4.5.1. FUNCIONES PARA CREAR MATRICES DISPERSAS (DIRECTORIO SPARFUN)

Las siguientes funciones permiten crear matrices dispersas. Casi todas estas funciones tienen muchas posibles formas de ser utilizadas, con distintos argumentos y valores de retorno. Se recomienda ver el ***Help*** de MATLAB para tener una información más detallada.

speye(m,n) Matriz identidad dispersa de tamaño $m \times n$ con unos en la diagonal
sprand(m,n) Matriz aleatoria dispersa con distribución uniforme
sprandn(m,n) Matriz aleatoria dispersa con distribución normal
sprandsym(n) Matriz aleatoria simétrica
spdiags(A) Matriz dispersa a partir de las diagonales de otra matriz

sparse(m,n) Crea una matriz dispersa de tamaño $m \times n$ con todos los elementos cero
sparse(A) Crea una matriz dispersa a partir de una matriz llena

sparse(i,j,val,m,n) Construye una matriz dispersa a partir de: **i** vector de índices de fila, **j** vector de índices de columna, **val** vector de valores, **m** número de filas, **n** número de columnas, y un 6º argumento que permite definir el máximo **nnz** (por defecto en el tamaño de **val**) por si se quieren añadir después más elementos
full(S) Convierte una matriz dispersa en una matriz llena
find(S) Encuentra los índices de los elementos distintos de cero y los devuelve como si la matriz fuera un vector (por columnas).
[i,j,val]=find(S) Devuelve índices de fila, de columna y valores de los elementos, a partir de los cuáles se puede volver a crear la matriz
spy(S) Representa en una figura los elementos distintos de cero de la matriz

nnz(S) Devuelve el número de elementos distintos de cero
nonzeros(S) Devuelve un vector lleno que contiene los elementos distintos de cero
nzmax(S) Memoria reservada para elementos distintos de cero
spones(S) Reemplazar los elementos distintos de cero por unos
spalloc(m,n,nzmax) Reserva espacio para una matriz dispersa $m \times n$
issparse(S) Devuelve *true* si el argumento es una matriz dispersa

4.5.2. OPERACIONES CON MATRICES DISPERSAS

Las matrices dispersas son más “delicadas” que las matrices llenas. En concreto, son muy sensibles a la ordenación de sus filas y columnas. El problema no es tanto la matriz dispersa en sí, como las matrices –también dispersas- que resultan de las factorizaciones LU o de Cholesky necesarias para resolver sistemas de ecuaciones, calcular valores y vectores propios, etc. En estas factorizaciones puede haber muchos elementos cero que dejan de serlo y esto es un grave problema para la eficiencia de los cálculos. Reordenando las filas y columnas de una matriz dispersa se puede minimizar el número de elementos que se hacen distintos de cero al factorizar (*llenado* o *fill-in*). MATLAB dispone de dos formas principales de reordenación: los métodos del mínimo grado (*minimum degree*) y de Cuthill-McKee inverso (*reversed Cuthill-McKee*). A continuación se describen las funciones más importantes de MATLAB en esta categoría.

spfun('fun', S) Aplica una función a los elementos distintos de cero de la matriz **S**
p=colmmd(S) Devuelve el vector de permutaciones de columnas calculado con el método del mínimo grado (minimum degree). Para matrices no simétricas esta permutación tiende a producir factorizaciones LU más dispersas.
p=symmmd(S) Devuelve el vector de permutaciones de filas y columnas (*symmetric minimum degree permutation*). Aplicando esta permutación a las filas y columnas se obtienen factorizaciones de Cholesky más dispersas.
p=symrcm(S) Obtiene un vector de permutaciones por el método de Cuthill-McKee inverso tal que, aplicado a filas y columnas de **S**, obtiene matrices con los

	elementos agrupados alrededor de la diagonal principal (mínima anchura de banda). Se aplica a matrices simétricas y no simétricas.
p=colperm(S)	Obtiene una permutación de columnas que ordena las columnas en orden de número de ceros no decreciente. A veces se utiliza para ordenar antes de aplicar la factorización LU. Si la matriz es simétrica la permutación se puede aplicar a filas y a columnas.
randperm(n)	Calcula una permutación aleatoria de los *n* primeros números naturales

4.5.3. OPERACIONES DE ÁLGEBRA LINEAL CON MATRICES DISPERSAS

A continuación se describen muy brevemente las funciones de MATLAB que pueden utilizarse para operar con matrices dispersas. Algunas de estas funciones se llaman igual que las correspondientes funciones para matrices llenas, y otras son específicas de matrices dispersas. Casi todas estas funciones tienen varias formas de utilizarse. Para más detalles se sugiere recurrir al ***Help***.

[L,U,P]=lu(S)	Realiza la factorización LU
L=chol(S)	Realiza la factorización de Cholesky
[Q,R]=qr(S)	Realiza la factorización QR
[L,U]=luinc(A,tol)	Realiza una factorización LU incompleta
L=cholinc(S)	Calcula una factorización de Cholesky incompleta (Ver la Ayuda)

[V,D,FLAG] = eigs(S)	Calcula algunos valores propios de una matriz cuadrada. Esta función tiene muchas posibles formas: consultar la Ayuda
svds(S)	Calcula algunos valores singulares de una matriz rectangular. Esta función tiene muchas posibles formas: consultar la Ayuda
normest(S,tol)	Estimación de la norma sub-2 con una determinada tolerancia (por defecto 1e-06)
condest(S)	Estimación de condición numérica sub-1
sprank(S)	Calcula el rango de una matriz dispersa
symbfact(S)	*Symbolic factorization analysis*. Devuelve información sobre los elementos que se harán distintos de cero en la factorización de Cholesky, sin llegar a realizar dicha factorización

Los sistemas de ecuaciones con matrices dispersas se pueden resolver con métodos directos, que son variantes de la eliminación gaussiana. El camino habitual de acceder a los métodos directos es a través de los operadores / y \, igual que para matrices llenas.

También se pueden utilizar métodos iterativos, que tienen la ventaja de no cambiar ningún elemento de la matriz. Se trata de obtener soluciones aproximadas después de un número finito de pasos.

Se llama factorizaciones "incompletas" a aquéllas que no calculan la factorización exacta sino una aproximada, despreciando los elementos que se hacen distintos de cero pero tienen un valor pequeño. Aunque la factorización es incompleta y sólo aproximada, se puede hacer en mucho menos tiempo y para ciertas finalidades es suficiente. Estas factorizaciones incompletas se utilizan por ejemplo como pre-condicionadores de algunos métodos iterativos.

Las siguientes funciones son muy especializadas y aquí sólo se van a citar sus nombres (en inglés, tal como los utiliza MATLAB). Para más información recurrir al ***Help*** y a la bibliografía especializada.

pcg()	Resuelve un sistema de ecuaciones lineales por el método del Gradiente Conjugado Pre-condicionado (Preconditioned Conjugate Gradients Method). La matriz debe ser simétrica y positivo-definida
bicg()	BiConjugate Gradients Method. Similar al anterior para matrices cuadradas que no son simétricas y positivo-definidas
bicgstab()	BiConjugate Gradients Stabilized Method.
cgs()	Conjugate Gradients Squared Method
gmres()	Generalized Minimum Residual Method
qmr()	Quasi-Minimal Residual Method
spparms()	Establece los parámetros para las funciones que trabajan con matrices sparse (set parameters for sparse matrix routines)
spaugment()	Form least squares augmented system

4.5.4. OPERACIONES CON MATRICES DISPERSAS

El criterio general para trabajar con matrices dispersas en MATLAB es que casi todas las operaciones matriciales estándar funcionan de la misma forma sobre matrices dispersas que sobre matrices llenas. De todas formas, existen algunos criterios particulares que conviene conocer y que se enuncian a continuación:

1. Las funciones que aceptan una matriz como argumento y devuelven un escalar o un vector siempre devuelven un vector lleno, aunque el argumento sea disperso
2. Las funciones que aceptan como argumentos escalares o vectores y devuelven matrices devuelven matrices llenas
3. Las funciones de un solo argumento que reciben una matriz y devuelven una matriz o vector conservan el carácter del argumento (disperso o lleno). Ej: ***chol()***, ***diag()***, ***max()***, ***sum()***
4. Las funciones binarias devuelven resultados dispersos si ambos argumentos son dispersos. Si un operando es lleno devuelven lleno, excepto si la operación conserva los elementos cero y distintos de cero (por ejemplo: .* y ./)
5. La concatenación de matrices con ***cat*** o corchetes [ ] produce resultados dispersos para operaciones mixtas
6. Sub-indexado de matrices; **S**(i,j) a la derecha de una asignación produce resultados dispersos, mientras que a la izquierda de una asignación (=) mantiene el tipo de almacenamiento de **S**.

4.5.5. PERMUTACIONES DE FILAS Y/O COLUMNAS EN MATRICES SPARSE

Para permutar las filas de una matriz se debe pre-multiplicar por una matriz de permutación **P**, que es una matriz que deriva de la matriz identidad **I** por permutación de filas y/o columnas. Así, el producto **P*****S** permuta filas de la matriz **S**, mientras que **S*****P'** permuta columnas.

Un vector de permutación **p** (que contiene una permutación de los números naturales *1:n*) actúa sobre las filas **S**(**p**,:) o columnas **S**(:,**p**). El vector de permutación **p** es más compacto y eficiente que la matriz de permutación **P**. Por eso casi siempre los resultados de permutaciones realizadas o a realizar se dan como vector **p** (excepto en la factorización LU). Las sentencias siguientes ilustran la relación entre la matriz **P** y el vector **p**.

```
» I = speye(5);
» p=[2,1,5,4,3]
p =
     2     1     5     4     3
```

```
P = I(p,:)       % para calcular la matriz P a partir del vector p
P =
   (2,1)        1
   (1,2)        1
   (5,3)        1
   (4,4)        1
   (3,5)        1
p = (P*(1:n)')' % para calcular el vector p a partir de la matriz P
p =
     2     1     5     4     3
```

Puede comprobarse que la inversa de **P** es **P'**. La función de reordenación ***symrcm(A)*** tiende a minimizar la banda de la matriz agrupando los elementos junto a la diagonal, y ***symmd(A)*** minimiza el fill-in o llenado de una matriz simétrica, mientras que ***colmmd(A)*** lo hace con una matriz no simétrica.

5. PROGRAMACIÓN DE MATLAB

Como ya se ha dicho varias veces –incluso con algún ejemplo– MATLAB es una aplicación que se puede programar muy fácilmente. De todas formas, como lenguaje de programación pronto verá que no tiene tantas posibilidades como otros lenguajes (ni tan complicadas...). Se comenzará viendo las bifurcaciones y bucles, y la lectura y escritura interactiva de variables, que son los elementos básicos de cualquier programa de una cierta complejidad.

5.1. Bifurcaciones y bucles

MATLAB posee un lenguaje de programación que –como cualquier otro lenguaje– dispone de sentencias para realizar ***bifurcaciones*** y ***bucles***. Las ***bifurcaciones*** permiten realizar una u otra operación según se cumpla o no una determinada condición. La Figura 13 muestra tres posibles formas de bifurcación.

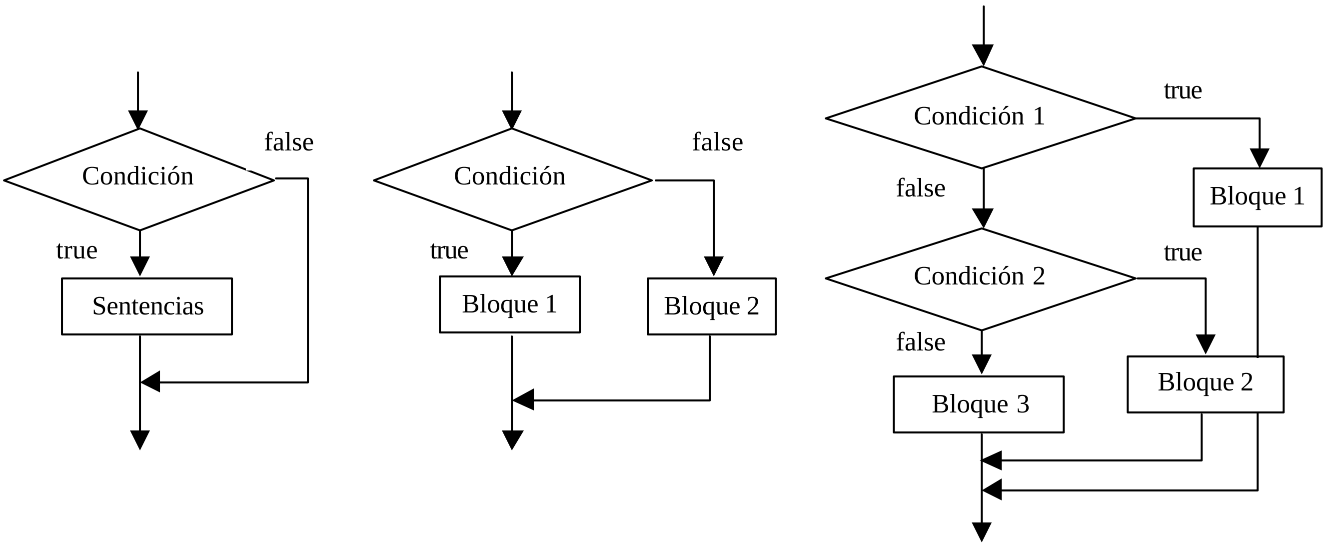


Figura 13. Ejemplos gráficos de bifurcaciones.

Los ***bucles*** permiten repetir las mismas o análogas operaciones sobre datos distintos. Mientras que en C/C++/Java el "cuerpo" de estas sentencias se determinaba mediante llaves {...}, en MATLAB se utiliza la palabra ***end*** con análoga finalidad. Existen también algunas otras diferencias de sintaxis.

La Figura 14 muestra dos posibles formas de bucle, con el control situado al principio o al final del mismo. Si el control está situado al comienzo del bucle es posible que las sentencias no se ejecuten ninguna vez, por no haberse cumplido la condición cuando se llega al bucle por primera vez. Sin embargo, si la condición está al final del bucle las sentencias se ejecutarán por lo menos una vez, aunque la condición no se cumpla. Muchos lenguajes de programación disponen de bucles con control al principio (***for*** y ***while*** en C/C++/Java) y al final (***do … while*** en C/C++/Java). En MATLAB no hay bucles con control al final del bucle, es decir, no existe la construcción análoga a ***do ... while***.

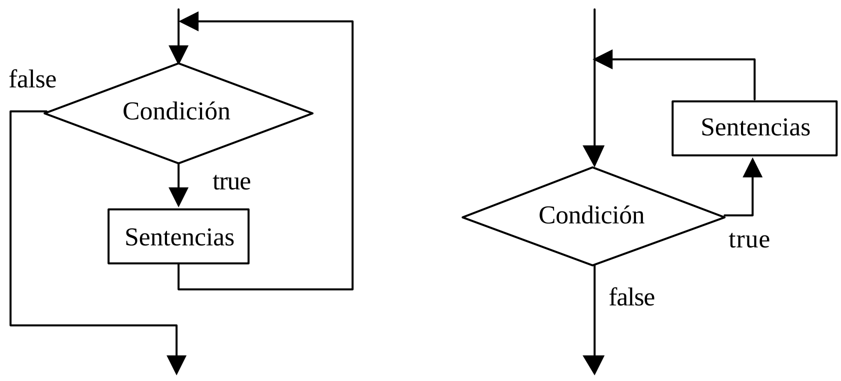


Figura 14. Bucles con control al principio y al final.

Las bifurcaciones y bucles no sólo son útiles en la preparación de programas o de ficheros ***.m**. También se aplican con frecuencia en el uso interactivo de MATLAB, como se verá más adelante en algunos ejemplos.

5.1.1. SENTENCIA *IF*

En su forma más simple, la sentencia ***if*** se escribe en la forma siguiente (obsérvese que –a diferencia de C/C++/Java– la condición no va entre paréntesis, aunque se pueden poner si se desea)[12]:

```
if condicion
    sentencias
end
```

Existe también la *bifurcación múltiple*, en la que pueden concatenarse tantas condiciones como se desee, y que tiene la forma:

```
if condicion1
    bloque1
elseif condicion2
    bloque2
elseif condicion3
    bloque3
else  % opción por defecto para cuando no se cumplan las condiciones 1,2,3
    bloque4
end
```

donde la opción por defecto ***else*** puede ser omitida: si no está presente no se hace nada en caso de que no se cumpla ninguna de las condiciones que se han chequeado.

Una observación muy importante: la condición del ***if*** puede ser una *condición matricial*, del tipo **A**==**B**, donde **A** y **B** son matrices del mismo tamaño. Para que se considere que la *condición* se cumple, es necesario que sean *iguales dos a dos **todos** los elementos* de las matrices **A** y **B** ($a_{ij}=b_{ij}$, $1\leq i\leq m$, $1\leq j\leq n$). Basta que haya dos elementos a_{ij} y b_{ij} diferentes para que las matrices ya no sean iguales, y por tanto las sentencias del ***if*** no se ejecuten. Análogamente, una condición en la forma A~=B exige que todos los elementos sean diferentes dos a dos (a_{ij}?b_{ij}, $1\leq i\leq m$, $1\leq j\leq n$). Bastaría que hubiera dos elementos a_{ij} y b_{ij} iguales para que la condición no se cumpliese. En resumen:

if A==B	exige que ***todos*** los elementos sean ***iguales*** dos a dos
if A~=B	exige que ***todos*** los elementos sean ***diferentes*** dos a dos

[12] En los ejemplos siguientes las ***sentencias*** aparecen desplazadas hacia la derecha respecto al ***if***, ***else*** o ***end***. Esto se hace así para que el programa resulte más legible, resultando más fácil ver dónde empieza y termina la bifurcación o el bucle. Es muy recomendable seguir esta práctica de programación.

Como se ha dicho, MATLAB dispone de funciones especiales para ayudar en el chequeo de condiciones matriciales. Por ejemplo, la función ***isequal(A, B)*** devuelve un uno si las dos matrices son idénticas y un cero en caso de que difieran en algo.

5.1.2. SENTENCIA *SWITCH*

La sentencia ***switch*** realiza una función análoga a un conjunto de ***if...elseif*** concatenados. Su forma general es la siguiente:

```
switch switch_expresion
    case case_expr1,
        bloque1
    case {case_expr2, case_expr3, case_expr4,...}
        bloque2
    ...
    otherwise,    % opción por defecto
        bloque3
end
```

Al principio se evalúa la ***switch_expresion***, cuyo resultado debe ser un número escalar o una cadena de caracteres. Este resultado se compara con las ***case_expr***, y se ejecuta el bloque de sentencias que corresponda con ese resultado. Si ninguno es igual a ***switch_expresion*** se ejecutan las sentencias correspondientes a ***otherwise***. Según puede verse en el ejemplo anterior, es posible agrupar varias condiciones dentro de unas llaves (constituyendo lo que se llama un *cell array* o vector de celdas, explicado en el Apartado 4.4); basta la igualdad con cualquier elemento del cell array para que se ejecute ese bloque de sentencias. La "igualdad" debe entenderse en el sentido del operador de igualdad (==) para escalares y la función ***strcmp()*** para cadenas de caracteres). A diferencia de C/C++/Java[13], en MATLAB sólo se ejecuta uno de los bloques relacionado con un ***case***.

5.1.3. SENTENCIA *FOR*

La sentencia ***for*** repite un conjunto de sentencias un número predeterminado de veces. La sentencia ***for*** de MATLAB es muy diferente y no tiene la generalidad de la sentencia ***for*** de C/C++/Java. La siguiente construcción ejecuta *sentencias* con valores de **i** de **1** a **n**, variando de uno en uno.

```
for i=1:n
    sentencias
end
```

o bien,

```
for i=vectorValores
    sentencias
end
```

donde **vectorValores** es un vector con los distintos valores que tomará la variable **i**.

En el siguiente ejemplo se presenta el caso más general para la variable del bucle (*valor_inicial: incremento: valor_final*); el bucle se ejecuta por primera vez con **i=n**, y luego **i** se va reduciendo de 0.2 en 0.2 hasta que llega a ser menor que 1, en cuyo caso el bucle se termina:

```
for i=n:-0.2:1
    sentencias
end
```

[13] En C se ejecuta el caso seleccionado y todos los siguientes, salvo que se utilice la sentencia ***break***.

En el siguiente ejemplo se presenta una estructura correspondiente a dos ***bucles anidados***. La variable **j** es la que varía más rápidamente (por cada valor de **i**, **j** toma todos sus posibles valores):

```
for i=1:m
    for j=1:n
        sentencias
    end
end
```

Una última forma de interés del bucle ***for*** es la siguiente (**A** es una matriz):

```
for i=A
    sentencias
end
```

en la que la variable **i** es un vector que va tomando en cada iteración el valor de una de las columnas de **A**.

Cuando se introducen interactivamente en la línea de comandos, los bucles ***for*** se ejecutan sólo después de introducir la sentencia ***end*** que los completa.

5.1.4. SENTENCIA *WHILE*

La estructura del bucle ***while*** es muy similar a la de C/C++/Java. Su sintaxis es la siguiente:

```
while condicion
    sentencias
end
```

donde ***condicion*** puede ser una expresión vectorial o matricial. Las *sentencias* se siguen ejecutando mientras haya elementos distintos de cero en ***condicion***, es decir, mientras haya algún o algunos elementos ***true***. El bucle se termina cuando *todos los elementos* de ***condicion*** son ***false*** (es decir, cero).

5.1.5. SENTENCIA *BREAK*

Al igual que en C/C++/Java, la sentencia ***break*** hace que se termine la ejecución del bucle más interno de los que comprenden a dicha sentencia.

5.1.6. SENTENCIAS *TRY...CATCH...END*

La construcción ***try...catch...end*** permite gestionar los errores que se pueden producir en tiempo de ejecución. Su forma es la siguiente:

```
try
    sentencias1
catch
    sentencias2
end
```

En el caso de que durante la ejecución del bloque ***sentencias1*** se produzca un error, el control de la ejecución se transfiere al bloque ***sentencias2***. Si la ejecución transcurriera normalmente, ***sentencias2*** no se ejecutaría nunca. MATLAB dispone de una función ***lasterr*** que devuelve una cadena de caracteres con el mensaje correspondiente al último error que se ha producido. En la forma ***lasterr('')*** pone a cero este contador de errores, y hace que la función ***lasterr*** devuelva la matriz vacía [] hasta que se produzca un nuevo error.

5.2. Lectura y escritura interactiva de variables

Se verá a continuación una forma sencilla de leer variables desde teclado y escribir mensajes en la pantalla del PC. Más adelante se considerarán otros modos más generales –y complejos– de hacerlo.

5.2.1. FUNCIÓN *INPUT*

La función ***input*** permite imprimir un mensaje en la línea de comandos de MATLAB y recuperar como valor de retorno un valor numérico o el resultado de una expresión tecleada por el usuario. Después de imprimir el mensaje, el programa espera que el usuario teclee el valor numérico o la expresión. Cualquier expresión válida de MATLAB es aceptada por este comando. El usuario puede teclear simplemente un vector o una matriz. En cualquier caso, la expresión introducida es evaluada con los valores actuales de las variables de MATLAB y el resultado se devuelve como valor de retorno. Véase un ejemplo de uso de esta función:

```
» n = input('Teclee el número de ecuaciones')
```

Otra posible forma de esta función es la siguiente (obsérvese el parámetro `'s'`):

```
» nombre = input('¿Cómo te llamas?','s')
```

En este caso el texto tecleado como respuesta se lee y se devuelve sin evaluar, con lo que se almacena en la cadena ***nombre***. Así pues, en este caso, si se teclea una fórmula, se almacena como texto sin evaluarse.

5.2.2. FUNCIÓN *DISP*

La función ***disp*** permite imprimir en pantalla un mensaje de texto o el valor de una matriz, pero sin imprimir su nombre. En realidad, ***disp*** siempre imprime vectores y/o matrices: las cadenas de caracteres son un caso particular de vectores. Considérense los siguientes ejemplos de cómo se utiliza:

```
» disp('El programa ha terminado')
» A=rand(4,4)
» disp(A)
```

Ejecútense las sentencias anteriores en MATLAB y obsérvese la diferencia entre las dos formas de imprimir la matriz **A**.

5.3. Ficheros *.m

Los ficheros con extensión (***.m***) son ficheros de texto sin formato (ficheros ASCII) que constituyen el centro de la programación en MATLAB. Ya se han utilizado en varias ocasiones. Estos ficheros se crean y modifican con un editor de textos cualquiera. En el caso de MATLAB 5.3 ejecutado en un PC bajo ***Windows***, lo más sencillo es utilizar su propio editor de textos.

Existen dos tipos de ficheros ****.m***, los ***ficheros de comandos*** (llamados *scripts* en inglés) y las ***funciones***. Los primeros contienen simplemente un conjunto de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero en la línea de comandos de MATLAB o se incluye dicho nombre en otro fichero ****.m***. Un fichero de comandos puede llamar a otros ficheros de comandos. Si un fichero de comandos se llama desde de la línea de comandos de MATLAB, las variables que crea pertenecen al ***espacio de trabajo base*** de MATLAB (recordar Apartado 1.4.4), y permanecen en él cuando se termina la ejecución de dicho fichero.

Las ***funciones*** permiten definir funciones enteramente análogas a las de MATLAB, con su ***nombre***, sus ***argumentos*** y sus ***valores de retorno***. Los ficheros ****.m*** que definen funciones permiten extender las posibilidades de MATLAB; de hecho existen bibliotecas de ficheros ****.m*** que

se venden (***toolkits***) o se distribuyen gratuitamente (a través de *Internet*). Las funciones definidas en ficheros ****.m*** se caracterizan porque la primera línea (que no sea un comentario) comienza por la palabra ***function***, seguida por los *valores de retorno* (entre corchetes [ ] y separados por comas, si hay más de uno), el signo igual (=) y el *nombre de la función*, seguido de los *argumentos* (entre paréntesis y separados por comas).

Recuérdese que un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo de forma recursiva. Los ficheros de comandos se pueden llamar también desde funciones, en cuyo caso las variables que se crean pertenecen a espacio de trabajo de la función. El espacio de trabajo de una función es independiente del espacio de trabajo base y del espacio de trabajo de las demás funciones. Esto implica por ejemplo que no puede haber colisiones entre nombres de variables: aunque varias funciones tengan una variable llamada A, en realidad se trata de variables completamente distintas (a no ser que **A** haya sido declarada como variable ***global***).

A continuación se verá con un poco más de detalle ambos tipos de ficheros ****.m***.

5.3.1. FICHEROS DE COMANDOS (*SCRIPTS*)

Como ya se ha dicho, los ficheros de comandos o *scripts* son ficheros con un nombre tal como ***file1.m*** que contienen una sucesión de comandos análoga a la que se teclearía en el uso interactivo del programa. Dichos comandos se ejecutan sucesivamente cuando se teclea el nombre del fichero que los contiene (sin la extensión), es decir cuando se teclea ***file1*** con el ejemplo considerado. Cuando se ejecuta desde la línea de comandos, las variables creadas por ***file1*** pertenecen al espacio de trabajo base de MATLAB. Por el contrario, si se ejecuta desde una función, las variables que crea pertenecen al espacio de trabajo de la función (ver Apartado 1.4.4, en la página 9).

En los ficheros de comandos conviene poner los puntos y coma (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso se puede llamar a sí mismo de modo recursivo. Sin embargo, no se puede hacer ***profile*** (ver Apartado 5.9, en la página 72) de un fichero de comandos: sólo se puede hacer de las funciones.

Las variables definidas por los ficheros de comandos son variables del espacio de trabajo desde el que se ejecuta el fichero, esto es variables con el mismo carácter que las que se crean interactivamente en MATLAB si el fichero se ha ejecutado desde la línea de comandos. Al terminar la ejecución del *script*, dichas variables permanecen en memoria.

El comando ***echo*** hace que se impriman los comandos que están en un *script* a medida que van siendo ejecutados. Este comando tiene varias formas:

echo on	activa el ***echo*** en todos los ficheros script
echo off	desactiva el ***echo***
echo file on	donde 'file' es el nombre de un fichero de función, activa el ***echo*** en esa función
echo file off	desactiva el ***echo*** en la función
echo file	pasa de **on** a **off** y viceversa
echo on all	activa el ***echo*** en todas las funciones
echo off all	desactiva el ***echo*** de todas las funciones

Mención especial merece el fichero de comandos ***startup.m*** (ver Apartado 1.4.2). Este fichero se ejecuta cada vez que se entra en MATLAB. En él puede introducir todos aquellos comandos que le interesa se ejecuten siempre al iniciar la sesión, por ejemplo ***format compact*** y los comandos necesarios para modificar el ***path***.

5.3.2. DEFINICIÓN DE FUNCIONES

La ***primera línea*** de un fichero llamado ***name.m*** que define una función tiene la forma:

```
function [lista de valores de retorno] = name(lista de argumentos)
```

donde ***name*** es el nombre de la función. Entre corchetes y separados por comas van los ***valores de retorno*** (siempre que haya más de uno), y entre paréntesis también separados por comas los ***argumentos***. Puede haber funciones sin valor de retorno y también sin argumentos. Recuérdese que los ***argumentos*** son los ***datos*** de la función y los ***valores de retorno*** sus ***resultados***. Si no hay valores de retorno se omiten los corchetes y el signo igual (=); si sólo hay un valor de retorno no hace falta poner corchetes. Tampoco hace falta poner paréntesis si no hay argumentos.

Una diferencia importante con C/C++/Java es que en MATLAB una función no modifica nunca los argumentos que recibe. Los resultados de una función de MATLAB se obtienen siempre a través de los valores de retorno, que pueden ser múltiples y matriciales. Tanto el número de argumentos como el de valores de retorno no tienen que ser fijos, dependiendo de cómo el usuario llama a la función[14].

Las variables definidas dentro de una función son ***variables locales***, en el sentido de que son inaccesibles desde otras partes del programa y en el de que no interfieren con variables del mismo nombre definidas en otras funciones o partes del programa. Se puede decir que pertenecen al propio espacio de trabajo de la función y no son vistas desde otros espacios de trabajo. Para que la función tenga acceso a variables que no han sido pasadas como argumentos es necesario declarar dichas variables como ***variables globales***, tanto en el programa principal como en las distintas funciones que deben acceder a su valor. Es frecuente utilizar el convenio de usar para las variables globales nombres largos (más de 5 letras) y con mayúsculas.

Por razones de eficiencia, los argumentos que recibe una función de MATLAB no se copian a variables locales si no son modificados por dicha función (en términos de C/C++ se diría que se pasan ***por referencia***). Esto tiene importantes consecuencias en términos de eficiencia y ahorro de tiempo de cálculo. Sin embargo, si dentro de la función se realizan modificaciones sobre los argumentos recibidos, antes se sacan copias de dichos argumentos a variables locales y se modifican las copias (diríase que en este caso los argumentos se pasan ***por valor***).

Dentro de la función, los valores de retorno deben ser calculados en algún momento (no hay sentencia ***return***, como en C/C++/Java). De todas formas, no hace falta calcular siempre todos los posibles valores de retorno de la función, sino sólo los que el usuario *espera obtener* en la sentencia de llamada a la función. En cualquier función existen dos variables definidas de modo automático, llamadas ***nargin*** y ***nargout***, que representan respectivamente el número de argumentos y el número de valores de retorno con los que la función ha sido llamada. Dentro de la función, estas variables pueden ser utilizadas como el programador desee.

La ejecución de una función termina cuando se llega a su última sentencia ejecutable. Si se quiere forzar el que una función termine de ejecutarse se puede utilizar la sentencia ***return***, que devuelve inmediatamente el control al entorno de llamada.

---

[14] Es un concepto distinto del de ***funciones sobrecargadas*** (funciones distintas con el mismo nombre y distintos argumentos), utilizadas en C/C++/Java. En MATLAB una misma función puede ser llamada con más o menos argumentos y valores de retorno. También en C/C++ es posible tener un número variable de argumentos, aunque no de valores de retorno.

5.3.3. FUNCIONES CON NÚMERO VARIABLE DE ARGUMENTOS

Desde la versión 5.0, MATLAB dispone de una nueva forma de pasar a una función un número variable de argumentos por medio de la variable ***varargin***, que es un ***vector de celdas*** (ver Apartado 4.4, en la página 44) que contienen tantos elementos como sean necesarios para poder recoger en dichos elementos todos los argumentos que se hayan pasado en la llamada. No es necesario que ***varargin*** sea el único argumento, pero sí debe ser el último, pues recoge todos los argumentos a partir de una determinada posición. Recuérdese que a los elementos de un ***cell array*** se accede utilizando llaves {}, en lugar de paréntesis ().

De forma análoga, una función puede tener un número indeterminado de valores de retorno utilizando ***varargout***, que es también un ***cell array*** que agrupa los últimos valores de retorno de la función. Puede haber otros valores de retorno, pero ***varargout*** debe ser el último. El cell array ***varargout*** se debe crear dentro de la función y hay que dar valor a sus elementos antes de salir de la función. Recuérdese también que las variables ***nargin*** y ***nargout*** indican el número de argumentos y de valores de retorno con que ha sido llamada la función. A continuación se presenta un ejemplo sencillo: obsérvese el código de la siguiente función ***atan3***:

```
function varargout=atan3(varargin)
   if nargin==1
      rad = atan(varargin{1});
   elseif nargin==2
      rad = atan2(varargin{1},varargin{2});
   else
      disp('Error: más de dos argumentos')
      return
   end
   varargout{1}=rad;
   if nargout>1
      varargout{2}=rad*180/pi;
   end
```

MATLAB (y muchos otros lenguajes de programación) dispone de dos funciones, llamadas ***atan*** y ***atan2***, para calcular el arco cuya tangente tiene un determinado valor. El resultado de dichas funciones está expresado en radianes. La función ***atan*** recibe un único argumento, con lo cual el arco que devuelve está comprendido entre $-\pi/2$ y $+\pi/2$ (entre –90º y 90º), porque por ejemplo un arco de 45º es indistinguible de otro de –135, si sólo se conoce la tangente. La función ***atan2*** recibe dos argumentos, uno proporcional al seno del ángulo y otro al coseno. En este caso ya se pueden distinguir los ángulos en los cuatro cuadrantes, entre $-\pi$ y π (entre –180º y 180º).

La función ***atan3*** definida anteriormente puede recibir uno o dos argumentos: si recibe uno llama a ***atan*** y si recibe dos llama a ***atan2*** (si recibe más da un mensaje de error). Además, ***atan3*** puede devolver uno o dos valores de retorno. Por ejemplo, si el usuario la llama en la forma:

```
» a = atan3(1);
```

devuelve un valor de retorno que es el ángulo en radianes, pero si se llama en la forma:

```
» [a, b] = atan3(1,-1);
```

devuelve dos valores de retorno, uno con el ángulo en radianes y otro en grados. Obsérvese cómo la función ***atan3*** utiliza los vectores de celdas ***varargin*** y ***varargout***, así como el número actual de argumentos ***nargin*** con los que ha sido llamada.

5.3.4. *HELP* PARA LAS FUNCIONES DE USUARIO

También las funciones creadas por el usuario pueden tener su ***help***, análogo al que tienen las propias funciones de MATLAB. Esto se consigue de la siguiente forma: las primeras líneas de

comentarios de cada fichero de función son muy importantes, pues permiten construir un ***help*** sobre esa función. En otras palabras, cuando se teclea en la ventana de comandos de MATLAB:

```
» help mi_func
```

el programa responde escribiendo las primeras líneas del fichero ***mi_func.m*** que comienzan por el carácter (%), es decir, que son comentarios.

De estas líneas, tiene una importancia particular la primera línea de comentarios (llamada en ocasiones línea H1). En ella hay que intentar poner la información más relevante sobre esa función. La razón es que existe una función, llamada ***lookfor*** que busca una determinada palabra en cada primera línea de comentario de todas las funciones ****.m***.

5.3.5. *HELP* DE DIRECTORIOS

MATLAB permite que los usuarios creen una ayuda general para todas las funciones que están en un determinado directorio. Para ello se debe crear en dicho directorio un fichero llamado ***contents.m***. A continuación se muestra un fichero típico ***contents.m*** correspondiente al directorio ***toolbox\local*** de MATLAB:

```
% Preferences.
%
% Saved preferences files.
%   startup     - User startup M-file.
%   finish      - User finish M-file.
%   matlabrc    - Master startup M-file.
%   pathdef     - Search path defaults.
%   docopt      - Web browser defaults.
%   printopt    - Printer defaults.
%
% Preference commands.
%   cedit       - Set command line editor keys.
%   terminal    - Set graphics terminal type.
%
% Configuration information.
%   hostid      - MATLAB server host identification number.
%   license     - License number.
%   version     - MATLAB version number.

% Utilities.
%   userpath    - User environment path.

% Copyright (c) 1984-98 by The MathWorks, Inc.
% $Revision: 1.10 $  $Date: 1998/05/28 19:55:36 $
```

Compruébese que la información anterior es exactamente la que se imprime con el comando

```
» help local
```

Si el fichero ***contents.m*** no existe, se listan las primeras líneas de comentarios (líneas H1) de todas las funciones que haya en ese directorio. Para que el ***Help*** de directorios funcione correctamente hace falta que ese directorio esté en el ***search path*** de MATLAB o que sea el directorio actual.

5.3.6. SUB-FUNCIONES

Tradicionalmente MATLAB obligaba a crear un fichero ****.m*** diferente por cada función. El nombre de la función debía coincidir con el nombre del fichero. A partir de la versión 5.0 se han introducido las ***sub-funciones***, que son funciones adicionales definidas en un mismo fichero ****.m***, con nombres diferentes del nombre del fichero (y del nombre de la función principal) y que las sub-funciones

sólo pueden ser llamadas por las funciones contenidas en ese fichero, resultando “invisibles” para otras funciones externas.

A continuación se muestra un ejemplo contenido en un fichero llamado ***mi_fun.m***:

```
function y=mi_fun(a,b)
y=subfun1(a,b);

function x=subfun1(y,z)
x=subfun2(y,z);

function x=subfun2(y,z)
x=y+z+2;
```

5.3.7. FUNCIONES PRIVADAS

Las funciones privadas (***private***) son funciones que no se pueden llamar desde cualquier otra función, aunque se encuentren en el ***path*** o en el directorio actual. Sólo ciertas funciones están autorizadas a utilizarlas. Las funciones privadas se definen en sub-directorios que se llaman ***private*** y sólo pueden ser llamadas por funciones definidas en el directorio padre del sub-directorio ***private***.

En la búsqueda de nombres que hace MATLAB cuando encuentra un nombre en una expresión, las funciones privadas se buscan inmediatamente después de las sub-funciones, y antes que las funciones de tipo general.

5.3.8. FUNCIONES *.P

Las funciones ****.p*** son funciones ****.m*** pre-compiladas con la función ***pcode***. Por defecto el resultado del comando ***pcode func.m*** es un fichero ***func.p*** en el directorio actual (el fichero ***func.m*** puede estar en cualquier directorio del ***search path***). El comando ***pcode -inplace func.m*** crea el fichero ***func.p*** en el mismo directorio donde encuentra el fichero ***func.m***. Pueden pasarse varios ficheros ****.m*** al comando ***pcode*** de una sola vez.

Los ficheros ****.p*** se ejecutan algo más rápidamente que los ****.m*** y permiten ocultar el código de los ficheros ASCII correspondientes a las funciones ****.m*** de MATLAB.

5.3.9. VARIABLES PERSISTENTES

Las ***variables persistentes*** son variables locales de las funciones (pertenecen al espacio de trabajo de la función y sólo son visibles en dicho espacio de trabajo), que ***conservan su valor*** entre distintas llamadas a la función. Por defecto, las variables locales de una función se crean y destruyen cada vez que se ejecuta la función. Las variables persistentes se pueden definir en funciones, pero no en ficheros de comandos. Es habitual utilizar para ellas letras mayúsculas. Las variables se declaran como persistentes utilizando la palabra ***persistent*** seguida de los nombres separados por blancos, como por ejemplo:

```
» persistent VELOCIDAD TIEMPO
```

Las variables ***persistent*** se inicializan a la matriz vacía [] y permanecen en memoria hasta que se hace ***clear*** de la función o cuando se modifica el *fichero-M*. Para evitar que un fichero-M se modifique se puede utilizar el comando ***mlock file.m***, que impide la modificación del fichero. El comando ***munlock*** desbloquea el fichero mientras que la función ***mislocked*** permite saber si está bloqueado o no.

5.3.10. VARIABLES GLOBALES

Las variables globales son visibles en todas las funciones (y en el espacio de trabajo base o general) que las declaran como tales. Dichas variables de declaran precedidas por la palabra ***global*** y separadas por blancos, en la forma:

```
global VARIABLE1 VARIABLE2
```

Como ya se ha apuntado, estas variables sólo son visibles en los espacios de trabajo de las funciones que las declaran como tales (y en el propio espacio de trabajo base, si también ahí han sido declaradas como globales). Ya se ha dicho también que se suele recurrir al criterio de utilizar nombres largos y con mayúsculas, para distinguirlas fácilmente de las demás variables.

5.4. Entrada y salida de datos

Ya se ha visto una forma de realizar la entrada interactiva de datos por medio de la función ***input*** y de imprimir resultados por medio de la función ***disp***. Ahora se van a ver otras formas de intercambiar datos con otras aplicaciones.

5.4.1. IMPORTAR DATOS DE OTRAS APLICACIONES

Hay varias formas de pasar datos de otras aplicaciones –por ejemplo de ***Excel***– a MATLAB. Se pueden enumerar las siguientes:

– se puede utilizar el ***Copy*** y ***Paste*** para copiar datos de la aplicación original y depositarlos entre los corchetes de una matriz o vector, en una línea de comandos de MATLAB. Tiene el inconveniente de que estos datos no se pueden editar.

– se puede crear un fichero ****.m*** con un editor de textos, con lo cual no existen problemas de edición.

– es posible leer un ***flat file*** escrito con caracteres ASCII. Un *flat file* es un fichero con filas de longitud constante separadas con ***Intro***, y varios datos por fila separados por *blancos*. Estos ficheros pueden ser leídos desde MATLAB con el comando ***load***. Si se ejecuta ***load datos.txt*** el contenido del ***flat file*** se deposita en una matriz con el nombre ***datos***. Por ejemplo, creando un fichero llamado ***flat.txt*** que contenga las líneas:

```
23.456  56.032 67.802
 3.749 -98.906 34.910
```

el comando `A=load('flat.txt')` leerá estos valores y los asignará a la matriz ***A***. Para más información utilizar ***help load***.

– el comando ***textread*** permite leer datos de cualquier tipo de un fichero siempre que estén convenientemente separados. Ver el ***Help*** para más información.

– se pueden leer datos de un fichero con las funciones ***fopen*** y ***fread*** (ver Apartados 5.5.1 y 5.5.3, en las páginas 62 y 63).

– existen también otros métodos posibles: escribir funciones en C para traducir a formato ****.mat*** (y cargar después con ***load***), crear un fichero ejecutable ****.mex*** que lea los datos, etc. No se verán en estos Apuntes.

5.4.2. EXPORTAR DATOS A OTRAS APLICACIONES

De forma análoga, también los resultados de MATLAB se pueden exportar a otras aplicaciones como ***Word*** o ***Excel***.

– utilizar el comando ***diary*** para datos de pequeño tamaño (ver Apartado 1.7, en la página 12)

– utilizar el comando ***save*** con la opción ***–ascii*** (ver Apartado 1.6, en la página 11)

– utilizar las funciones de bajo nivel ***fopen***, ***fwrite*** y otras (ver Apartados 5.5.1 y 5.5.3, en las páginas 62 y 63)

– otros métodos que no se verán aquí: escribir subrutinas en C para traducir de formato ****.mat*** (guardando previamente con ***save***), crear un fichero ejecutable ****.mex*** que escriba los datos, etc.

Hay que señalar que los ficheros binarios ****.mat*** son trasportables entre versiones de MATLAB en distintos tipos de computadores, porque contienen información sobre el tipo de máquina en el *header* del fichero, y el programa realiza la transformación de modo automático. Los ficheros ****.m*** son de tipo ASCII, y por tanto pueden ser leídos por distintos computadores sin problemas de ningún tipo.

5.5. Lectura y escritura de ficheros

MATLAB dispone de funciones de lectura/escritura análogas a las del lenguaje C (en las que están inspiradas), aunque con algunas diferencias. En general son versiones simplificadas –con menos opciones y posibilidades– que las correspondientes funciones de C.

5.5.1. FUNCIONES *FOPEN* Y *FCLOSE*

Estas funciones sirven para abrir y cerrar ficheros, respectivamente. La función ***fopen*** tiene la forma siguiente:

```
[fi,texto] = fopen('filename','c')
```

donde **fi** es un valor de retorno que sirve como identificador del fichero, **texto** es un mensaje para caso de que se produzca un error, y **c** es un carácter (o dos) que indica el tipo de operación que se desea realizar. Las opciones más importantes son las siguientes:

'r'	lectura (de *read*)
'w'	escritura reemplazando (de *write*)
'a'	escritura a continuación (de *append*)
'r+'	lectura y escritura

Cuando por alguna razón el fichero no puede ser abierto, se devuelve un (-1). En este caso el valor de retorno **texto** puede proporcionar información sobre el tipo de error que se ha producido (también existe una función llamada ***ferror*** que permite obtener información sobre los errores. En el ***Help*** del programa se puede ver cómo utilizar esta función).

Después de realizar las operaciones de lectura y escritura deseadas, el fichero se puede cerrar con la función ***close*** en la forma siguiente:

```
st = fclose(fi)
```

donde **st** es un valor de retorno para posibles condiciones de error. Si se quieren cerrar a la vez todos los ficheros abiertos puede utilizarse el comando:

```
st = close('all')
```

5.5.2. FUNCIONES *FSCANF*, *SSCANF*, *FPRINTF* Y *SPRINTF*

Estas funciones permiten leer y escribir en ficheros ASCII, es decir, en ficheros formateados. La forma general de la función ***fscanf*** es la siguiente:

```
[var1,var2,...] = fscanf(fi,'cadena de control',size)
```

donde **fi** es el identificador del fichero (devuelto por la función ***fopen***), y **size** es un argumento opcional que puede indicar el tamaño del vector o matriz a leer. Obsérvese otra diferencia con C: las variables leídas se devuelven como valor de retorno y no como argumentos pasados por referencia (precedidos por el carácter &). La ***cadena de control*** va encerrada entre apóstrofos simples, y contiene los especificadores de formato para las variables:

%s	para cadenas de caracteres
%d	para variables enteras
%f	para variables de punto flotante
%lf	para variables de doble precisión

La función ***sscanf*** es similar a ***fscanf*** pero la entrada de caracteres no proviene de un fichero sino de una cadena de caracteres.

Finalmente, la función ***fprintf*** dirige su salida formateada hacia el fichero indicado por el identificador. Su forma general es:

```
fprintf(fi,'cadena de control',var1,var2,...)
```

Esta es la función más parecida a su homóloga de C. La cadena de control contiene los formatos de escritura, que son similares a los de C, como muestran los ejemplos siguientes:

```
fprintf(fi,'El número de ecuaciones es: %d\n',n)
fprintf(fi,'El determinante es: %lf10.4\n',n)
```

De forma análoga, la función ***sprintf*** convierte su resultado en una cadena de caracteres que devuelve como valor de retorno, en vez de enviarlo a un fichero. Véase un ejemplo:

```
resultado = sprintf('El cuadrado de %f es %12.4f\n',n,n*n)
```

donde **resultado** es una cadena de caracteres. Esta función constituye el método más general de convertir números en cadenas de caracteres, por ejemplo para ponerlos como títulos de figuras.

5.5.3. FUNCIONES *FREAD* Y *FWRITE*

Estas funciones son análogas a ***fscanf*** y ***fprintf***, pero en vez de leer o escribir en un fichero de texto (ASCII), lo hacen en un *fichero binario*, no legible directamente por el usuario. Aunque dichos ficheros no se pueden leer y/o modificar con un editor de textos, tienen la ventaja de que las operaciones de lectura y escritura son mucho más rápidas, eficientes y precisas (no se pierden decimales al escribir). Esto es particularmente significativo para grandes ficheros de datos. Para más información sobre estas funciones se puede utilizar el ***help***.

5.5.4. FICHEROS DE ACCESO DIRECTO

De ordinario los ficheros de disco se leen y escriben secuencialmente, es decir, de principio a final, sin volver nunca hacia atrás ni realizar saltos. Sin embargo, a veces interesa acceder a un fichero de un modo arbitrario, sin ningún orden preestablecido. Esto se puede conseguir con las funciones ***ftell*** y ***fseek***.

En cada momento, hay una especie de *cursor* que indica en qué parte del fichero se está posicionado. La función ***fseek*** permite mover este cursor hacia delante o hacia atrás, respecto a la posición actual ('cof'), respecto al principio ('bof') o respecto al final del fichero ('eof'). La función ***ftell*** indica en qué posición está el cursor. Si alguna vez se necesita utilizar este tipo de acceso a disco, se puede buscar más información por medio del ***help***.

5.6. Recomendaciones generales de programación

Las funciones vectoriales de MATLAB son mucho más rápidas que sus contrapartidas escalares. En la medida de lo posible es muy interesante vectorizar los algoritmos de cálculo, es decir, realizarlos con vectores y matrices, y no con variables escalares dentro de bucles.

Aunque los vectores y matrices pueden ir creciendo a medida que se necesita, es mucho más rápido reservarles toda la memoria necesaria al comienzo del programa. Se puede utilizar para ello la función ***zeros***. Además de este modo la memoria reservada es contigua.

Es importante utilizar el ***profile*** para conocer en qué sentencias de cada función se gasta la mayor parte del tiempo de cálculo. De esta forma se descubren "cuellos de botella" y se pueden desaroollar aplicaciones mucho más eficientes.

Conviene desarrollar los programas incrementalmente, comprobando cada función o componente que se añade. De esta forma siempre se construye sobre algo que ya ha sido comprobado y que funciona: si aparece algún error, lo más probable es que se deba a lo último que se ha añadido, y de esta manera la búsqueda de errores está acotada y es mucho más sencilla. Recuérdese que de ordinario el tiempo de corrección de errores en un programa puede ser 4 ó 5 veces superior al tiempo de programación. El ***debugger*** es una herramienta muy útil a la hora de acortar ese tiempo de puesta a punto.

En este mismo sentido, puede decirse que pensar bien las cosas al programar (sobre una hoja de papel en blanco, mejor que sobre la pantalla del PC) siempre es rentable, porque se disminuye más que proporcionalmente el tiempo de depuración y eliminación de errores.

Otro objetivo de la programación debe ser mantener el código lo más sencillo y ordenado posible. Al pensar en cómo hacer un programa o en cómo realizar determinada tarea es conveniente pensar siempre primero en la solución más sencilla, y luego plantearse otras cuestiones como la eficiencia.

Finalmente, el código debe ser escrito de una manera clara y ordenada, introduciendo comentarios, utilizando líneas en blanco para separar las distintas partes del programa, sangrando las líneas para ver claramente el rango de las bifurcaciones y bucles, utilizando nombres de variables que recuerden al significado de la magnitud física correspondientes, etc.

En cualquier caso, la mejor forma (y la única) de aprender a programar es programando.

5.7. Llamada a comandos del sistema operativo y a otras funciones externas

Estando en la ventana de comandos de MATLAB, se pueden ejecutar comandos de MS-DOS precediéndolos por el carácter (!)

```
» !edit fichero1.m
```

Si el comando va seguido por el carácter ampersand (&) el comando se ejecuta en "background", es decir, se recupera el control del programa sin esperar que el comando termine de ejecutarse. Por ejemplo, para arrancar ***Notepad*** en background,

```
» !notepad &
```

Existe también la posibilidad de arrancar una aplicación y dejarla iconizada. Esto se hace postponiendo el carácter barra vertical (|), como por ejemplo en el comando:

```
» !notepad |
```

Algunos comandos de MATLAB realizan la misma función que los comandos análogos del sistema operativo MS-DOS, con lo que se puede evitar utilizar el operador (!). Algunos de estos comandos son los siguientes:

dir	contenido del directorio actual
what	ficheros ***.m** en el directorio actual
delete filename	borra el fichero llamado ***filename***
mkdir(nd)	crea un sub-directorio con el nombre ***nd***
copyfile(sc, dst)	copia el fichero ***sc*** en el fichero ***dst***
type file.txt	imprime por la pantalla el contenido del fichero de texto ***file.txt***
cd	cambiar de directorio activo
pwd	muestra el path del directorio actual
which func	localiza una función llamada ***func***
lookfor palabra	busca ***palabra*** en todas las primeras líneas de los ficheros ***.m**

5.8. Funciones de función

En MATLAB existen funciones a las que hay que pasar como argumento el nombre de otras funciones, para que puedan ser llamadas desde dicha función. Así sucede por ejemplo si se desea calcular la integral definida de una función, resolver una ecuación no lineal, o integrar numéricamente una ecuación diferencial ordinaria (problema de valor inicial). Estos serán los tres casos –de gran importancia práctica– que se van a ver a continuación. Se comenzará por medio de un ejemplo, utilizando una función llamada ***prueba*** que se va a definir en un fichero llamado ***prueba.m***.

Para definir esta función, se debe elegir ***FILE/New/M-File*** en el menú de MATLAB. Si las cosas están "en orden" se abrirá el ***Editor&Debugger*** para que se pueda editar ese fichero. Una vez abierto el ***Editor***, se deben teclear las 2 líneas siguientes:

```
function y=prueba(x)
y = 1./((x-.3).^2+.01)+1./((x-.9).^2+.04)-6;
```

salvándolo después con el nombre de ***prueba.m***. La definición de funciones se ha visto con detalle en el Apartado 5.3.2, a partir de la página 57. El fichero anterior ha definido una nueva función que puede ser utilizada como cualquier otra de las funciones de MATLAB. Antes de seguir adelante, conviene ver el aspecto que tiene esta función que se acaba de crear. Para dibujar la función ***prueba***, tecléense los siguientes comandos:

```
» x=-1:0.1:2;
» plot(x,prueba(x))
```

El resultado aparece en la Figura 15. Ya se está en condiciones de intentar hacer cálculos y pruebas con esta función.

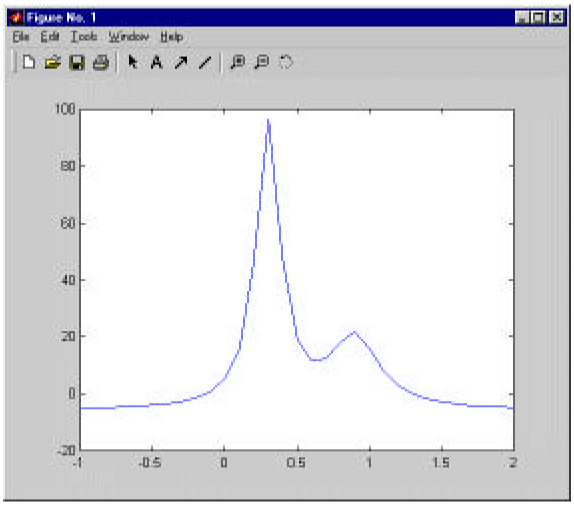


Figura 15. Función “prueba”.

5.8.1. INTEGRACIÓN NUMÉRICA DE FUNCIONES

Lo primero que se va a hacer es calcular la integral definida de esta función entre dos valores de la abscisa **x**. En inglés, al cálculo numérico de integrales definidas se le llama *quadrature*. Sabiendo eso, no resulta extraño el comando con el cual se calcula el área comprendida bajo la función entre

los puntos 0 y 1 (obsérvese que el nombre de la función a integrar se pasa entre apóstrofos, como cadena de caracteres):

```
» area = quad('prueba', 0, 1)
area =
   29.8583
```

Si se teclea ***help quad*** se puede obtener más de información sobre esta función, incluyendo el método utilizado (*Simpson*) y la forma de controlar el error de la integración.

La función ***quad8()*** utiliza un método de orden superior (*Newton-Cotes*), mientras que la función ***dblquad()*** realiza integrales definidas dobles. Ver el ***Help*** o los manuales on-line para más información.

5.8.2. ECUACIONES NO LINEALES Y OPTIMIZACIÓN

Después de todo, calcular integrales definidas no es tan difícil. Más difícil es desde luego calcular las raíces de ecuaciones no lineales, y el mínimo o los mínimos de una función. MATLAB dispone de las tres funciones siguientes:

fzero	calcula un cero o una raíz de una función de una variable
fminbnd	calcula el mínimo de una función de una variable
fminsearch	calcula el mínimo de una función de varias variables
optimset	permite establecer los parámetros del proceso de cálculo

Se empezará con el cálculo de raíces. Del gráfico de la función ***prueba*** entre -1 y 2 resulta evidente que dicha función tiene dos raíces en ese intervalo. La función ***fzero*** calcula una y se conforma: ¿Cuál es la que calcula? Pues depende de un parámetro o argumento que indica un punto de partida para buscar la raíz. Véanse los siguientes comandos y resultados:

```
» fzero('prueba', -.5)
ans =
   -0.1316
» fzero('prueba', 2)
ans =
    1.2995
```

En el primer caso se ha dicho al programa que empiece a buscar en el punto -0.5 y la solución encontrada ha sido -0.1316. En el segundo caso ha empezado a buscar en el punto de abscisa 2 y ha encontrado otra raíz en el punto 1.2995. Se ven claras las limitaciones de esta función.

La función ***fzero()*** tiene también otras formas interesantes:

fzero('prueba', [x1,x2])	calcula una raíz en el intervalo x1-x2. Es necesario que la función tenga distinto signo en los extremos del intervalo.
fzero('prueba', x, options)	calcula la raíz más próxima a **x** con ciertas opciones definidas en la estructura **options**. Esta estructura se crea con la función ***optimset***.

La función ***optimset*** tiene la siguientes formas generales:

```
options = optimset('param1',val1,'param2',val2,...
```

en la que se indican los nombres de los parámetros u opciones que se desean modificar y los valores que se desea dar para cada uno de dichos parámetros.

```
options = optimset(oldopts, 'param1',val1,'param2',val2,...)
```

en la que se obtienen unas nuevas opciones modificando unas opciones anteriores con una serie de parejas *nombre-valor* de parámetros.

Existen muchas opciones que pueden ser definidas por medio de la función ***optimset***. Algunas de las más características son las siguientes (las dos primeras están dirigidas a evitar procesos iterativos que no acaben nunca y la tercera a controlar la precisión en los cálculos):

MaxFunEvals	máximo número de evaluaciones de función permitidas
MaxIter	máximo número de iteraciones
TolX	error máximo permitido en la abscisa de la raíz

Ahora se va a calcular el mínimo de la función ***prueba***. Defínase una función llamada ***prueba2*** que sea ***prueba*** cambiada de signo, y trátese de reproducir en el PC los siguientes comandos y resultados (para calcular máximos con ***fmin*** bastaría con cambiar el signo de la función):

```
» plot(x,prueba2(x))
» fminbnd('prueba2', -1,2)
Optimization terminated successfully:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000e-004

ans =
    0.3004
» fminbnd('prueba2', 0.5,1)
Optimization terminated successfully:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000e-004

ans =
    0.8927
```

También a la función ***fminbnd*** se le puede pasar la estructura ***options***. Por ejemplo, para fijar un error de 10^{-08} se puede proceder del siguiente modo:

```
» options=optimset('TolX', 1e-08);
» fminbnd('prueba2', 0.5,1, options)
```

En cualquier caso, es importante observar que para calcular las raíces o los valores mínimos de una función, hay que pasar el nombre de esta función como argumento a la función de MATLAB que va a hacer los cálculos. En esto consiste el concepto de ***función de función***.

MATLAB tiene un *toolbox* o paquete especial (no disponible en las Salas de PCs la Escuela) con muchas más funciones orientadas a la *optimización*, es decir al cálculo de valores mínimos de funciones, con o sin restricciones.

5.8.3. INTEGRACIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES ORDINARIAS

Este es otro campo en el que las capacidades de MATLAB pueden resultar de gran utilidad. MATLAB es capaz de calcular la evolución en el tiempo de sistemas de ecuaciones diferenciales ordinarias de primer orden, lineales y no lineales. Por el momento se supondrá que las ecuaciones diferenciales se pueden escribir en la forma:

$$\dot{\mathbf{y}} = \mathbf{f}\left(\mathbf{y}, t\right) \tag{7}$$

donde t es la variable escalar, y tanto $\mathbf{y}$ como su derivada son vectores. Un ejemplo típico puede ser el ***tiro parabólico***, considerando una resistencia del aire proporcional al cuadrado de la velocidad. Se supone que dicha fuerza responde a la expresión vectorial:

$$\begin{Bmatrix} F_x \\ F_y \end{Bmatrix} = -c\sqrt{\left(\dot{x}^2 + \dot{y}^2\right)}\begin{Bmatrix} \dot{x} \\ \dot{y} \end{Bmatrix} \tag{8}$$

donde c es una constante conocida. Las ecuaciones diferenciales del movimiento serán:

$$\begin{Bmatrix}\ddot{x}\\ \ddot{y}\end{Bmatrix} = \begin{Bmatrix}0\\ -g\end{Bmatrix} - \frac{c}{m}\sqrt{\left(\dot{x}^2+\dot{y}^2\right)}\begin{Bmatrix}\dot{x}\\ \dot{y}\end{Bmatrix} \qquad (9)$$

pero éste es un sistema de 2 ecuaciones diferenciales de orden 2. Para poderlo integrar debe tener la forma del sistema (7), y para ello se va a trasformar en 4 ecuaciones de primer orden, de la forma siguiente:

$$\begin{Bmatrix}\dot{u}\\ \dot{v}\\ \dot{x}\\ \dot{y}\end{Bmatrix} = \begin{Bmatrix}0\\ -g\\ u\\ v\end{Bmatrix} - \frac{c}{m}\sqrt{\left(u^2+v^2\right)}\begin{Bmatrix}u\\ v\\ 0\\ 0\end{Bmatrix} \qquad (10)$$

MATLAB dispone de varias funciones para integrar sistemas de ecuaciones diferenciales ordinarias de primer orden, entre ellas ***ode23***, que utiliza el método de *Runge-Kutta* de segundo/tercer orden, y ***ode45***, que utiliza el método de *Runge-Kutta-Fehlberg* de cuarto/quinto orden. Ambas exigen al usuario escribir una función que calcule las derivadas a partir del vector de variables, en la forma indicada por la ecuación (7).

Cree con el ***Editor/Debugger*** un fichero llamado ***tiropar.m*** que contenga las siguientes líneas:

```
function deriv=tiropar(t,y)
fac=-(0.001/1.0)*sqrt((y(1)^2+y(2)^2));
deriv=zeros(4,1);
deriv(1)=fac*y(1);
deriv(2)=fac*y(2)-9.8;
deriv(3)=y(1);
deriv(4)=y(2);
```

donde se han supuesto unas constantes con los valores de c=0.001, m=1 y g=9.8. Falta fijar los valores iniciales de posición y velocidad. Se supondrá que el proyectil parte del origen con una velocidad de 100 m/seg y con un ángulo de 30º, lo que conduce a los valores iniciales siguientes: u(0)=100*cos(pi/6), v(0)=100*sin(pi/6), x(0)=0, y(0)=0. Los comandos para realizar la integración son los siguientes (se podrían agrupar en un fichero llamado ***tiroparMain.m***):

```
» t0=0; tf=9;
» y0=[100*cos(pi/6) 100*sin(pi/6) 0 0]';
» [t,Y]=ode23('tiropar',[t0,tf],y0);
» plot(t,Y(:,4))     % dibujo de la altura en función del tiempo
```

donde [t0,tf] es un vector que define el intervalo temporal de integración. Es muy importante que en la función ***ode23***, el vector de condiciones iniciales **y0** sea un vector columna. El vector **t** devuelto por ***ode23*** contiene los valores del tiempo para los cuales se ha calculado la posición y velocidad. Dichos valores son controlados por la función ***ode23*** y no por el usuario, por lo que de ordinario no estarán igualmente espaciados. La matriz de resultados **Y** contiene cuatro columnas (las dos velocidades y las dos coordenadas de cada posición) y tantas filas como elementos tiene el vector **t**. En la Figura 16 se muestra el resultado del ejemplo anterior (posición vertical en función del tiempo).

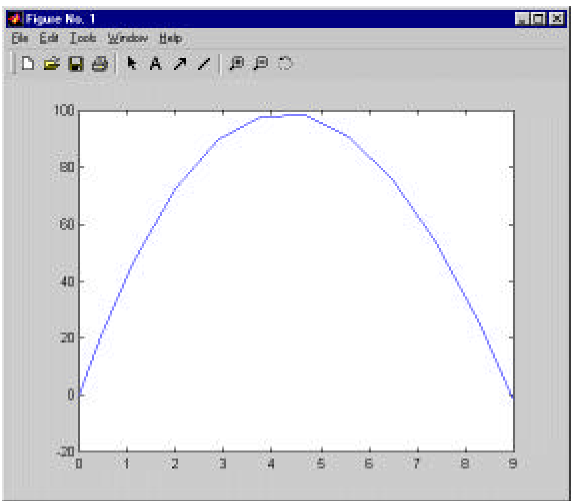


Figura 16. Tiro parabólico (posición vertical en función del tiempo).

MATLAB dispone de varias funciones para la integración de sistemas de ecuaciones diferenciales ordinarias. Se pueden citar las siguientes, clasificadas según una característica de las ecuaciones que se desea integrar:

Sistemas no-rígidos	ode23, lode45 y ode113
Sistemas rígidos	ode15s, ode23s, odq23t y ode23tb

La ***rigidez*** (*stiffness*, en la literatura inglesa) es una característica de muchos sistemas de ecuaciones diferenciales ordinarias que aparecen en la práctica y que los hace más difíciles de resolver. Una explicación detallada de esta característica excede la finalidad de este manual, pero sí se puede dar una muy breve explicación.

Muchos integradores numéricos están basados en fórmulas que permiten predecir el valor de la función en **t+**D**t** a partir del valor de la función y de su derivada en el instante **t** y anteriores:

$$\mathbf{y}_{t+\Delta t} = \overline{f}\left(\mathbf{y}_t, \mathbf{y}_{t-\Delta t}, \ldots, \dot{\mathbf{y}}_t, \dot{\mathbf{y}}_{t-\Delta t}, \ldots, t\right) \tag{11}$$

A estos integradores se les llama ***integradores explícitos***. Todo lo que necesitan es que el usuario programe una función que calcule la derivada en la forma indicada en la ecuación (7).

En la solución de un sistema de ecuaciones diferenciales ordinarias aparecen combinadas diversas componentes oscilatorias (tipo seno, coseno o similar). Algunas de estas componentes oscilan más rápidamente que otras (tienen una frecuencia más elevada). Los problemas ***rígidos*** o ***stiff*** son aquellos en cuya solución participan componentes de frecuencias muy diferentes (muy altas y muy bajas). Todos los integradores de MATLAB tienen control automático del error. Quiere esto decir que el usuario fija el error que está dispuesto a admitir en la solución y MATLAB ajusta el paso de la integración para conseguir ese error. Los integradores explícitos detectan la posible presencia de componentes de alta frecuencia en la solución y tratan de adaptar a ellas su paso, que se hace demasiado pequeño y termina por detener la integración.

Los ***integradores implícitos*** son mucho más apropiados para los problemas ***stiff***. En lugar de utilizar fórmulas del tipo de la ecuación (11) utilizan fórmulas del tipo:

$$\mathbf{y}_{t+\Delta t} = \overline{f}\left(\mathbf{y}_{t+\Delta t}, \mathbf{y}_t, \mathbf{y}_{t-\Delta t}, \ldots, \dot{\mathbf{y}}_{t+\Delta t}, \dot{\mathbf{y}}_t, \dot{\mathbf{y}}_{t-\Delta t}, \ldots, t\right) \tag{12}$$

El problema con la expresión (12) es que para calcular la función en **t+**D**t** hace uso de la derivada en ese mismo instante, que no puede ser conocida si no se conoce la función. Eso quiere decir que el sistema (12) es un sistema de ecuaciones no lineal que hay que resolver iterativamente. Los sistemas de ecuaciones no lineales se resuelven mucho más rápidamente si se conoce la derivada de la función (un ejemplo es el método de *Newton-Raphson*). Los integradores ***stiff*** de MATLAB son capaces de calcular esta derivada numéricamente (por diferencias finitas), pero son mucho más eficientes si el usuario es capaz de escribir una segunda función que les dé esta derivada. A esta derivada, que en realidad es una matriz de derivadas, se le suele llamar ***Jacobiano***. Los integradores ***stiff***, además de la ecuación (7), permiten para el sistema de ecuaciones diferenciales una forma algo más especializada:

$$\mathbf{M}(\mathbf{y}, t)\dot{\mathbf{y}} - \mathbf{f}(\mathbf{y}, t) = \mathbf{0} \tag{13}$$

en cuyo caso el usuario también tiene que proporcionar una función que calcule la matriz **M(y,***t***)**. La ecuación (13) representa una gran número de casos prácticos, por ejemplo los que surgen de las ecuaciones diferenciales del movimiento en Mecánica.

La forma más básica para todos los integradores de MATLAB es la siguiente:

```
[t, Y] = solvername('F', tspan, y0)
```

donde **F** es el nombre de la función que permite calcular la derivada según la expresión (7), **tspan** puede ser un vector de dos elementos [**tini, tfinal**] que representan el comienzo y el fin de la integración o un vector de tiempos en los cuales se desea que MATLAB devuelva resultados, e **y0** es un vector columna con los valores iniciales. Como resultado se obtiene el vector **t** de tiempos en los que se dan resultados y una matriz **Y** con tantas filas como tiempos de salida y que representan cada una de ellas la salida en el correspondiente instante de tiempo.

Una forma más elaborada de llamar a los integradores de MATLAB es la siguiente:

```
[t, Y, s] = solvername('F', tspan, y0, options)
```

donde **s** es un vector con ciertos resultados estadísticos de la integración (ver el ***Help*** para más detalle) y ***options*** es una estructura similar a la vista en el Apartado anterior para el cálculo de raíces y mínimos. En este caso la estructura ***options*** (que es diferente de la anterior, aunque se esté utilizando el mismo nombre) se determina por medio de la función ***odeset***, que admite las formas:

```
options = odeset('param1', val1,'param2', val2, ...);
options = odeset(oldopt, 'param1', val1,'param2', val2, ...);
```

Entre los parámetros u opciones más importantes se pueden citar los siguientes:

Para el error	RelTol , AbsTol
Para el paso	InitialStep, MaxStep
Para la matriz M	Mass, MassSingular
Para el Jacobiano	Jacobian, JConstant, JPattern, Vectorized
Para la salida	OutputFcn, OutputSel, Refine, Stats

A continuación se va a repetir el ejemplo de tiro parabólico presentado al comienzo de esta Sección utilizando el integrador implícito ***ode15s*** con algunas opciones modificadas. Para ello la ecuación (10) se va a re-escribir en la forma de la ecuación (13), resultando:

$$\begin{bmatrix} m & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \dot{u} \\ \dot{v} \\ \dot{x} \\ \dot{y} \end{Bmatrix} = \begin{Bmatrix} 0 \\ -mg \\ u \\ v \end{Bmatrix} - c\sqrt{\left(u^2+v^2\right)} \begin{Bmatrix} u \\ v \\ 0 \\ 0 \end{Bmatrix} \qquad (14)$$

En este caso el programa principal se ha denominado ***tiroparMain2*** y tiene la siguiente forma:

```
t0=0; tf=10; npoints=51;
y0=[100*cos(pi/6),100*sin(pi/6),0,0]';
% vector de puntos en los que se desea resultados
tspan=[t0:(tf-t0)/(npoints-1):tf];
% modificación de las opciones por defecto
options = odeset('RelTol',1e-06,'AbsTol',1e-06,'Stats','on', 'Mass','M(t,y)');
% llamada a la función de integración numérica
[t,Y]=ode15s('tiropar2',tspan,y0,options);
% dibujo de la altura del móvil en función del tiempo
plot(t,Y(:,4))
```

Obsérvese cómo se han definido nuevas tolerancias para los errores absoluto y relativo, que se ha activado la opción de imprimir estadísticas y que se le indica al programa que se le va a dar una matriz de masas que depende de la posición y del tiempo (en este caso no es así, pero de este modo se presenta el ejemplo de una forma más general. Las otras opciones para el argumentos **Mass**' son '**M**' y '**M(t)**'). La función ***tiropar2*** ha sufrido cambios importantes respecto a ***tiropar*** y tiene la siguiente forma:

```
function varargout=tiropar2(t,y,str)
m=1;
deriv=zeros(4,1);
M=eye(4); M(1,1)=m; M(2,2)=m;

switch str
   case ''
      fac=-(0.001)*sqrt((y(1)^2+y(2)^2));
      deriv(1)=fac*y(1);
      deriv(2)=fac*y(2)-9.8*m;
      deriv(3)=y(1);
      deriv(4)=y(2);
      varargout{1}=deriv;
   case 'mass'
      varargout{1}=M;
end
```

El cambio más importante consiste en que la función ***tiropar2*** se utiliza tanto para dar el valor de la función **F** como de la matriz **M** en la expresión (13). Esto se consigue definiendo un tercer argumento **arg** tal que cuando se omite la función devuelve **F** y cuando vale '***mass***' la función devuelve la matriz de masas. Éste es el convenio utilizado por los integradores de MATLAB. En esta función se han utilizado las técnicas de número variable de argumentos y valores de retorno explicadas en el Apartado 5.3.3, en la página 58.

El resultado de MATLAB incluye las estadísticas solicitadas y es el siguiente:

```
58 successful steps
2 failed attempts
97 function evaluations
1 partial derivatives
14 LU decompositions
90 solutions of linear systems
```

No se puede entrar con más detenimiento en estas cuestiones especializadas. Para el lector interesado en estos problemas se recomienda acudir a la ayuda de MATLAB en formato PDF, concretamente al ***Capítulo 8*** del manual ***Using MATLAB***, accesible desde la ventana principal de MATLAB por medio del comando ***Help/Help Desk (HTML)*** y luego desde el enlace ***Online Manuals (in PDF)***, que se encuentra en la parte inferior izquierda de la página. Estos manuales contienen una explicación muy detallada de todas las posibilidades de las funciones referidas, así como numerosos ejemplos.

5.8.4. LAS FUNCIONES *EVAL*, *EVALC*, *FEVAL Y EVALIN*

Estas funciones tienen mucho que ver con las cadenas de caracteres, ya que necesitan la flexibilidad de éstas para alcanzar todas sus posibilidades. Las funciones para manipular cadenas de caracteres se verán en un próximo apartado.

La función ***eval***('cadena de caracteres') hace que se evalúe como expresión de MATLAB el texto contenido entre las comillas como argumento de la función. Este texto puede ser un comando, una fórmula matemática o -en general- cualquier expresión válida de MATLAB. La función ***eval*** debe tener los valores de retorno necesarios para recoger los resultados de la expresión evaluada.

Esta forma de definir ***macros*** es particularmente útil para pasar nombres de función a otras funciones definidas en ficheros ****.m***.

El siguiente ejemplo va creando variables llamadas **A1**, **A2**, ..., **A10** utilizando la posibilidad de concatenar cadenas antes de pasárselas como argumento a la función ***eval***:

```
for n = 1:10
    eval(['A',num2str(n),' = magic(n)'])
end
```

La función ***eval()*** se puede usar también en la forma ***eval('tryString', 'catchString')***. En este caso se evalúa la cadena ***'tryString'***, y si se produce algún error se evalúa la cadena ***'catchString'***. Es una forma simplificada de gestionar errores en tiempo de ejecución.

La función ***T=evalc()*** es similar a ***eval()*** pero con la diferencia de que cualquier salida que la expresión pasada como argumento hubiera enviado a la ventana de comandos de MATLAB es capturada, almacenada en una matriz de caracteres **T** cuyas filas terminan con el carácter '\n'.

Por su parte la función ***feval*** sirve para evaluar, dentro de una función, otra función cuyo nombre está contenido en una cadena de caracteres. Es posible que este nombre se haya leído desde teclado o se haya recibido como argumento. A la función ***feval*** hay que pasarle como argumentos tanto el nombre de la función a evaluar como sus argumentos. Por ejemplo, si dentro de una función se quiere evaluar la función ***calcular(A, b, c)***, donde el nombre ***calcular*** se envía como argumento en la cadena ***nombre***, entonces ***feval(nombre, A, b, c)*** equivale a ***calcular(A, b, c)***.

Finalmente, la función ***evalin(ws, 'expresion')*** evalúa *'expresion'* en el espacio de trabajo ***ws***. Los dos posibles valores para ***ws*** son ***'caller'*** y ***'base'***, que indican el espacio de trabajo de la función que llama a ***evalin*** o el espacio de trabajo base. Los valores de retorno se pueden recoger del modo habitual.

5.9. Distribución del esfuerzo de cálculo: *Profiler*

El ***profiler*** es una utilidad que permite saber qué tiempo de cálculo se ha gastado en cada línea de una función definida en un fichero ****.m*** o en general de un programa de MATLAB. Permite asimismo determinar el número de llamadas a dicha función, las funciones que la han llamado (*parent functions*), las funciones llamadas por ella (*child functions*), etc.

El ***profiler*** mejora la calidad de los programas, pues permite detectar los “cuellos de botella” de una aplicación y concentrar en ellos los esfuerzos para mejorar su eficiencia. Por ejemplo, sabiendo el número de veces que se llama a una función y el tiempo que cuesta cada llamada, se puede decidir si es mejor emplear más memoria en guardar resultados intermedios para no tener que calcular varias veces lo mismo.

El ***profiler*** ha sido muy mejorado en la versión 5.3 de MATLAB. Se puede medir el tiempo (en centésimas de segundo) empleado en cada línea del fichero, en cada llamada a una función e incluso en cada operador del lenguaje. Una forma de llamar al ***profiler*** podría ser la siguiente (se supone que estas líneas forman parte de un fichero *.m):

```
profile on -detail operator;
[T, Y] = ode113('RTDyn2m', tspan, y0, myOptions);
profile report;
```

donde con la primera línea se activa el ***profiler*** a la vez que se define el ***grado de detalle*** que se desea. La segunda línea es una llamada a la función ***ode113*** que a su vez llama a muchas otras funciones y la tercera línea detiene el ***profiler*** y le pide que genere un informe en HTML con los resultados calculados. La Figura 17 muestra el aspecto de la página principal del informe, con información sobre todas las funciones y operadores; la Figura 18 muestra el detalle del informa sobre una función particular.

Existen tres posibles grados de detalle respecto a la información que se le pide al ***profiler***:

mmex — determina el tiempo utilizado por funciones y sub-funciones definidas en ficheros ****.m*** y ****.mex***. Ésta es la opción por defecto.

builtin	como el anterior pero incluyendo las ***funciones intrínsecas*** de MATLAB.
operator	como ***builtin*** pero incluyendo también el tiempo empleado por los ***operadores*** tales como la suma + y el producto *.

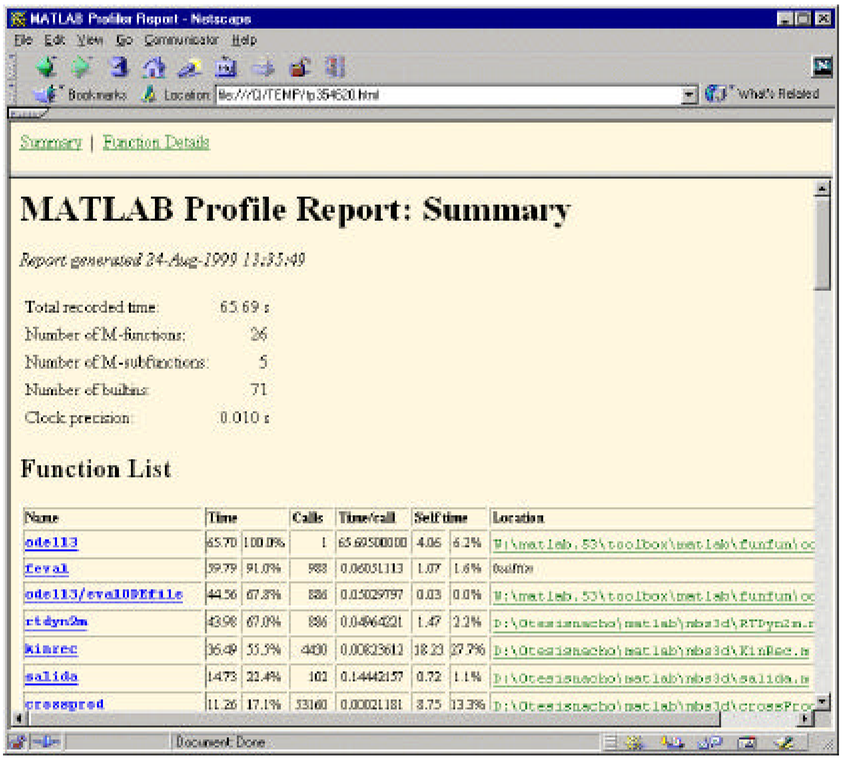


Figura 17. Página principal del informe del profiler.

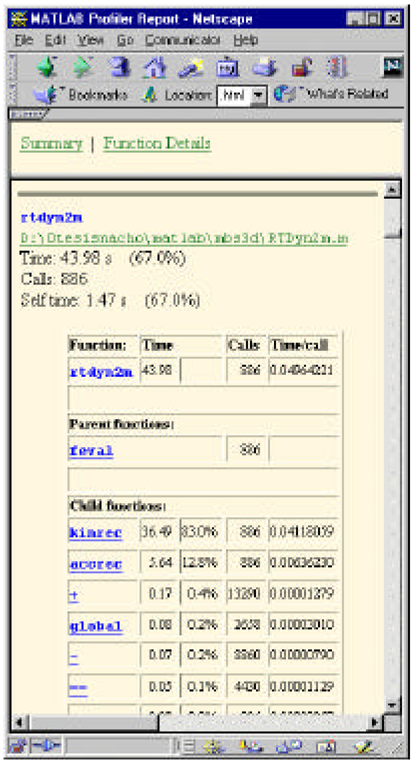


Figura 18. Informe sobre una función.

Otros posibles comandos relacionados con el ***profiler*** de MATLAB son los siguientes:

profile on	activa el ***profiler*** poniendo a cero los contadores
profile on –detail level	como el anterior, pero con el grado de detalle indicado
profile on –history	activa el ***profiler*** guardando información sobre el orden de las llamadas
profile off	desactiva el ***profiler*** sin poner a cero los contadores
profile resume	vuelve a activar el ***profiler*** sin poner a cero los contadores
profile clear	pone a cero los contadores
profile report	detiene el ***profiler***, genera páginas HTML con los resultados y los muestra en un browser
profile report basename	genera un informe consistente en varios ficheros HTML en el directorio actual; los nombre de los ficheros están basados en el nombre ***basename***, que debe darse sin extensión
profile plot	detiene el ***profiler*** y representa gráficamente los resultados en un diagrama de barras correspondientes a las funciones más usadas
profile status	muestra una estructura conteniendo los datos del ***profile***
stats = profile('info')	detiene el ***profiler*** y muestra una estructura con los resultados

El ***profiler*** sólo se puede aplicar a funciones, no a ficheros de comandos. Para más información se puede consultar la ayuda del programa.

6. GRÁFICOS BIDIMENSIONALES

A estas alturas, después de ver cómo funciona este programa, a nadie le puede resultar extraño que los gráficos 2-D de MATLAB estén fundamentalmente orientados a la representación gráfica de vectores (y matrices). En el caso más sencillo los argumentos básicos de la función ***plot*** van a ser vectores. Cuando una matriz aparezca como argumento, se considerará como un conjunto de vectores columna (en algunos casos también de vectores fila).

MATLAB utiliza un tipo especial de ventanas para realizar las operaciones gráficas. Ciertos comandos abren una ventana nueva y otros dibujan sobre la ventana activa, bien sustituyendo lo que hubiera en ella, bien añadiendo nuevos elementos gráficos a un dibujo anterior. Todo esto se verá con más detalle en las siguientes secciones.

6.1. Funciones gráficas 2D elementales

MATLAB dispone de cuatro funciones básicas para crear gráficos 2-D. Estas funciones se diferencian principalmente por el *tipo de escala* que utilizan en los ejes de abscisas y de ordenadas. Estas cuatro funciones son las siguientes:

plot()	crea un gráfico a partir de vectores y/o columnas de matrices, con escalas lineales sobre ambos ejes.
loglog()	ídem con escala logarítmica en ambos ejes
semilogx()	ídem con escala lineal en el eje de ordenadas y logarítmica en el eje de abscisas
semilogy()	ídem con escala lineal en el eje de abscisas y logarítmica en el eje de ordenadas

En lo sucesivo se hará referencia casi exclusiva a la primera de estas funciones (***plot***). Las demás se pueden utilizar de un modo similar.

Existen además otras funciones orientadas a añadir títulos al gráfico, a cada uno de los ejes, a dibujar una cuadrícula auxiliar, a introducir texto, etc. Estas funciones son las siguientes:

title('título')	añade un título al dibujo
xlabel('tal')	añade una etiqueta al eje de abscisas. Con ***xlabel off*** desaparece
ylabel('cual')	añade una etiqueta al eje de ordenadas. Con ***ylabel off*** desaparece
text(x,y,'texto')	introduce 'texto' en el lugar especificado por las coordenadas **x** e **y**. Si **x** e **y** son vectores, el texto se repite por cada par de elementos. Si **texto** es también un vector de cadenas de texto de la misma dimensión, cada elemento se escribe en las coordenadas correspondientes
gtext('texto')	introduce **texto** con ayuda del ratón: el cursor cambia de forma y se espera un clic para introducir el texto en esa posición
legend()	define rótulos para las distintas líneas o ejes utilizados en la figura. Para más detalle, consultar el ***Help***
grid	activa la inclusión de una cuadrícula en el dibujo. Con ***grid off*** desaparece la cuadrícula

Borrar texto (u otros elementos gráficos) es un poco más complicado; de hecho, hay que preverlo de antemano. Para poder hacerlo hay que recuperar previamente el *valor de retorno* del comando con el cual se ha creado. Después hay que llamar a la función ***delete*** con ese valor como argumento. Considérese el siguiente ejemplo:

```
» v = text(1,.0,'seno')
v =
   76.0001
» delete(v)
```

Los dos grupos de funciones anteriores no actúan de la misma forma. Así, la función ***plot*** dibuja una nueva figura en la ventana activa (en todo momento MATLAB tiene una ventana activa de entre todos las ventanas gráficas abiertas), o abre una nueva figura si no hay ninguna abierta, sustituyendo cualquier cosa que hubiera dibujada anteriormente en esa ventana. Para verlo, se comenzará creando un par de vectores **x** e **y** con los que trabajar:

```
» x=[-10:0.2:10]; y=sin(x);
```

Ahora se deben ejecutar los comandos siguientes (se comienza cerrando la ventana activa, para que al crear la nueva ventana aparezca en primer plano):

```
» close          % se cierra la ventana gráfica activa anterior
» grid           % se crea una ventana con una cuadrícula
» plot(x,y)      % se dibuja la función seno borrando la cuadrícula
```

Se puede observar la diferencia con la secuencia que sigue:

```
» close
» plot(x,y)      % se crea una ventana y se dibuja la función seno
» grid           % se añade la cuadrícula sin borrar la función seno
```

En el primer caso MATLAB ha creado la cuadrícula en una ventana nueva y luego la ha borrado al ejecutar la función ***plot***. En el segundo caso, primero ha dibujado la función y luego ha añadido la cuadrícula. Esto es así porque hay funciones como ***plot*** que por defecto crean una nueva figura, y otras funciones como ***grid*** que se aplican a la ventana activa modificándola, y sólo crean una ventana nueva cuando no existe ninguna ya creada. Más adelante se verá que con la función ***hold*** pueden añadirse gráficos a una figura ya existente respetando su contenido.

6.1.1. FUNCIÓN *PLOT*

Esta es la función clave de todos los gráficos 2-D en MATLAB. Ya se ha dicho que el elemento básico de los gráficos bidimensionales es el **vector**. Se utilizan también cadenas de 1, 2 ó 3 caracteres para indicar *colores* y *tipos de línea*. La función ***plot()***, en sus diversas variantes, no hace otra cosa que dibujar vectores. Un ejemplo muy sencillo de esta función, en el que se le pasa un único vector como argumento, es el siguiente:

```
» x=[1 3 2 4 5 3]
x =
     1     3     2     4     5     3
» plot(x)
```

El resultado de este comando es que se abre una ventana mostrando el gráfico de la Figura 19. Por defecto, los distintos puntos del gráfico se unen con una línea continua. También por defecto, el color que se utiliza para la primera línea es el azul.

Cuando a la función ***plot()*** se le pasa un único vector –real– como argumento, dicha función dibuja en ordenadas el valor de los **n** elementos del vector frente a los índices 1, 2, ... **n** del mismo en abscisas. Más adelante se verá que si el vector es complejo, el funcionamiento es bastante diferente.

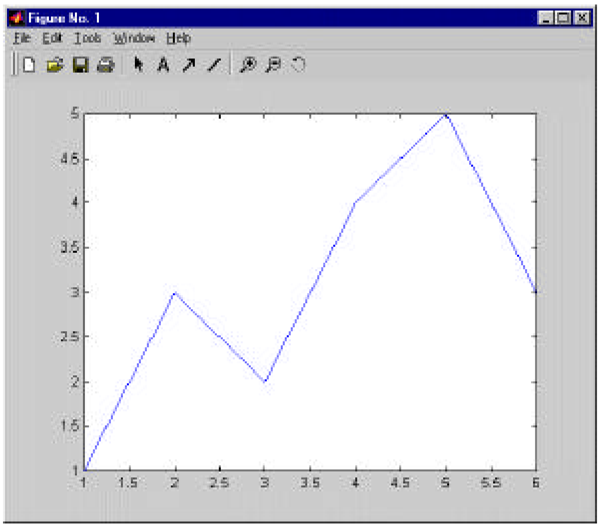


Figura 19. Gráfico del vector x=[1 3 2 4 5 3].

En la pantalla de su ordenador se habrá visto que MATLAB utiliza por defecto color blanco para el fondo de la pantalla y otros colores más oscuros para los ejes y las gráficas.

Una segunda forma de utilizar la función ***plot()*** es con dos vectores como argumentos. En este caso los elementos del segundo vector se representan en ordenadas frente a los valores del primero, que se representan en abscisas. Véase por ejemplo cómo se puede dibujar un cuadrilátero de esta forma (obsérvese que para dibujar un polígono cerrado el último punto debe coincidir con el primero):

```
» x=[1 6 5 2 1]; y=[1 0 4 3 1];
» plot(x,y)
```

La función ***plot()*** permite también dibujar múltiples curvas introduciendo varias parejas de vectores como argumentos. En este caso, cada uno de los segundos vectores se dibujan en ordenadas como función de los valores del primer vector de la pareja, que se representan en abscisas. Si el usuario no decide otra cosa, para las sucesivas líneas se utilizan colores que son permutaciones cíclicas del ***azul***, ***verde***, ***rojo***, ***cyan***, ***magenta***, ***amarillo*** y ***negro***. Obsérvese bien cómo se dibujan el seno y el coseno en el siguiente ejemplo:

```
» x=0:pi/25:6*pi;
» y=sin(x); z=cos(x);
» plot(x,y,x,z)
```

Ahora se va a ver lo que pasa con los **vectores complejos**. Si se pasan a ***plot()*** varios vectores complejos como argumentos, MATLAB simplemente representa las partes reales y desprecia las partes imaginarias. Sin embargo, un único argumento complejo hace que se represente la parte real en abscisas, frente a la parte imaginaria en ordenadas. Véase el siguiente ejemplo. Para generar un vector complejo se utilizará el resultado del cálculo de valores propios de una matriz formada aleatoriamente:

```
» plot(eig(rand(20,20)),'+')
```

donde se ha hecho uso de elementos que se verán en la siguiente sección, respecto a dibujar con distintos tipos de “markers” (en este caso con signos +), en vez de con línea continua, que es la opción por defecto. En el comando anterior, el segundo argumento es un carácter que indica el tipo de marker elegido. El comando anterior es equivalente a:

```
» z=eig(rand(20,20));
» plot(real(z),imag(z),'+')
```

Como ya se ha dicho, si se incluye más de un vector complejo como argumento, se ignoran las partes imaginarias. Si se quiere dibujar varios vectores complejos, hay que separar explícitamente las partes reales e imaginarias de cada vector, como se acaba de hacer en el último ejemplo.

El comando ***plot*** puede utilizarse también con matrices como argumentos. Véanse algunos ejemplos sencillos:

plot(A)	dibuja una línea por cada columna de A en ordenadas, frente al índice de los elementos en abscisas
plot(x,A)	dibuja las columnas (o filas) de **A** en ordenadas frente al vector **x** en abscisas. Las dimensiones de **A** y **x** deben ser coherentes: si la matriz **A** es cuadrada se dibujan las columnas, pero si no lo es y la dimensión de las filas coincide con la de **x**, se dibujan las filas
plot(A,x)	análogo al anterior, pero dibujando las columnas (o filas) de **A** en abscisas, frente al valor de **x** en ordenadas

plot(A,B)	dibuja las columnas de **B** en ordenadas frente a las columnas de **A** en abscisas, dos a dos. Las dimensiones deben coincidir
plot(A,B,C,D)	análogo al anterior para cada par de matrices. Las dimensiones de cada par deben coincidir, aunque pueden ser diferentes de las dimensiones de los demás pares

Se puede obtener una excelente y breve descripción de la función ***plot()*** con el comando ***help plot*** o ***helpwin plot***. La descripción que se acaba de presentar se completará en la siguiente sección, en donde se verá cómo elegir los colores y los tipos de línea.

6.1.2. ESTILOS DE LÍNEA Y MARCADORES EN LA FUNCIÓN *PLOT*

En la sección anterior se ha visto cómo la tarea fundamental de la función ***plot()*** era dibujar los valores de un vector en ordenadas, frente a los valores de otro vector en abscisas. En el caso general esto exige que se pasen como argumentos un par de vectores. En realidad, el conjunto básico de argumentos de esta función es una *tripleta* formada por dos vectores y una cadena de 1, 2 ó 3 caracteres que indica el color y el tipo de línea o de marker. En la tabla siguiente se pueden observar las distintas posibilidades.

Símbolo	Color	Símbolo	Marcadores (markers)
y	yellow	.	puntos
m	magenta	o	círculos
c	cyan	x	marcas en x
r	red	+	marcas en +
g	green	*	marcas en *
b	blue	s	marcas cuadradas (square)
w	white	d	marcas en diamante (diamond)
k	black	^	triángulo apuntando arriba
		v	triángulo apuntando abajo
Símbolo	**Estilo de línea**	>	triángulo apuntando a la dcha
-	líneas continuas	<	triángulo apuntando a la izda
:	líneas a puntos	p	estrella de 5 puntas
-.	líneas a barra-punto	h	estrella se seis puntas
--	líneas a trazos		

Tabla 1. Colores, markers y estilos de línea.

Cuando hay que dibujar varias líneas, por defecto se van cogiendo sucesivamente los colores de la tabla comenzando por el azul, hacia arriba, y cuando se terminan se vuelve a empezar otra vez por el azul. Si el fondo es blanco, este color no se utiliza para las líneas.

6.1.3. AÑADIR LÍNEAS A UN GRÁFICO YA EXISTENTE

Existe la posibilidad de añadir líneas a un gráfico ya existente, sin destruirlo o sin abrir una nueva ventana. Se utilizan para ello los comandos ***hold on*** y ***hold off***. El primero de ellos hace que los gráficos sucesivos respeten los que ya se han dibujado en la figura (es posible que haya que modificar la escala de los ejes); el comando ***hold off*** deshace el efecto de ***hold on***. El siguiente ejemplo muestra cómo se añaden las gráficas de **x2** y **x3** a la gráfica de **x** previamente creada (cada una con un tipo de línea diferente):

```
» plot(x)
» hold on
» plot(x2,'--')
» plot(x3,'-.')
» hold off
```

6.1.4. COMANDO *SUBPLOT*

Una ventana gráfica se puede dividir en **m** particiones horizontales y **n** verticales, con objeto de representar múltiples gráficos en ella. Cada una de estas subventanas tiene sus propios ejes, aunque otras propiedades son comunes a toda la figura. La forma general de este comando es:

```
subplot(m,n,i)
```

donde **m** y **n** son el número de subdivisiones en filas y columnas, e **i** es la subdivisión que se convierte en activa. Las subdivisiones se numeran consecutivamente empezando por las de la primera fila, siguiendo por las de la segunda, etc. Por ejemplo, la siguiente secuencia de comandos genera cuatro gráficos en la misma ventana:

```
» y=sin(x); z=cos(x); w=exp(-x*.1).*y; v=y.*z;
» subplot(2,2,1), plot(x,y)
» subplot(2,2,2), plot(x,z)
» subplot(2,2,3), plot(x,w)
» subplot(2,2,4), plot(x,v)
```

Se puede practicar con este ejemplo añadiendo títulos a cada *subplot*, así como rótulos para los ejes. Se puede intentar también cambiar los tipos de línea. Para volver a la opción por defecto basta teclear el comando:

```
» subplot(1,1,1)
```

6.1.5. CONTROL DE LOS EJES

También en este punto MATLAB tiene sus opciones por defecto, que en algunas ocasiones puede interesar cambiar. El comando básico es el comando ***axis***. Por defecto, MATLAB ajusta la escala de cada uno de los ejes de modo que varíe entre el mínimo y el máximo valor de los vectores a representar. Este es el llamado modo "auto", o modo automático. Para definir de modo explícito los valores máximo y mínimo según cada eje, se utiliza el comando:

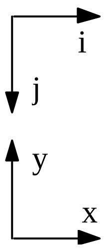
```
axis([xmin, xmax, ymin, ymax])
```

mientras que :

```
axis('auto')
```

devuelve el escalado de los ejes al valor por defecto o automático. Otros posibles usos de este comando son los siguientes:

v=axis	devuelve un vector **v** con los valores [xmin, xmax, ymin, ymax]
axis(axis)	mantiene los ejes en sus actuales valores, de cara a posibles nuevas gráficas añadidas con ***hold on***
axis('ij')	utiliza *ejes de pantalla*, con el origen en la esquina superior izda. y el eje **j** en dirección vertical descendente
axis('xy')	utiliza *ejes cartesianos* normales, con el origen en la esquina inferior izda. y el eje **y** vertical ascendente
axis('equal')	el escalado es igual en ambos ejes
axis('square')	la ventana será cuadrada



axis('image')	la ventana tendrá las proporciones de la imagen que se desea representar en ella (por ejemplo la de una imagen bitmap que se desee importar) y el escalado de los ejes será coherente con dicha imagen
axis('normal')	elimina las restricciones introducidas por 'equal' y 'square'
axis('off')	elimina las etiquetas, los números y los ejes
axis('on')	restituye las etiquetas, los números y los ejes

6.1.6. Función *line()*

La función line() permite dibujar una o más líneas que unen los puntos cuyas coordenadas se pasan como argumentos. Permite además especificar el color, grosor, tipo de trazo, marcador, etc. Es una función de más bajo nivel que la función plot(), pero ofrece una mayor flexibilidad. En su versión más básica, para dibujar un segmento de color verde entre dos puntos, esta función se llamaría de la siguiente manera:

```
line([xini, xend]', [yini, yend]', 'color', 'g')
```

Se puede también dibujar dos líneas a la vez utilizando la forma:

```
line([xini1 xini2; xend1 xend2], ([yini1 yini2; yend1 yend2]);
```

Finalmente, si cada columna de la matriz **X** contiene la coordenada *x* inicial y final de un punto, y lo mismo las columnas de la matriz **Y** con las coordenadas *y*, la siguiente sentencia dibuja tantas líneas como columnas tengan las matrices **X** e **Y**:

```
line([X], [Y]);
```

Se pueden controlar las características de la línea por medio de pares parámetro/valor, como por ejemplo:

```
line(x,y,'Color','r','LineWidth',4,'MarkerSize',12,'LineStyle','—','Marker','*')
```

6.2. Control de ventanas gráficas: Función *figure*

Si se llama a la función ***figure*** sin argumentos, se crea una nueva ventana gráfica con el número consecutivo que le corresponda. El valor de retorno es dicho número.

Por otra parte, el comando ***figure(n)*** hace que la ventana **n** pase a ser la ventana o figura activa. Si dicha ventana no existe, se crea una nueva ventana con el número consecutivo que le corresponda (que se puede obtener como valor de retorno del comando). La función ***close*** cierra la figura activa, mientras que ***close(n)*** cierra la ventana o figura número **n**.

El comando ***clf*** elimina el contenido de la figura activa, es decir, la deja abierta pero vacía. La función ***gcf*** devuelve el número de la figura activa en ese momento.

Para practicar un poco con todo lo que se acaba de explicar, ejecútense las siguientes instrucciones de MATLAB, observando con cuidado los efectos de cada una de ellas en la ventana activa. El comando ***figure(gcf)*** (*get current figure*) permite hacer visible la ventana de gráficos desde la ventana de comandos.

```
» x=[-4*pi:pi/20:4*pi];
» plot(x,sin(x),'r',x,cos(x),'g')
» title('Función seno(x) -en rojo- y función coseno(x) -en verde-')
» xlabel('ángulo en radianes'), figure(gcf)
» ylabel('valor de la función trigonométrica'), figure(gcf)
» axis([-12,12,-1.5,1.5]), figure(gcf)
» axis('equal'), figure(gcf)
» axis('normal'), figure(gcf)
» axis('square'), figure(gcf)
```

```
» axis('off'), figure(gcf)
» axis('on'), figure(gcf)
» axis('on'), grid, figure(gcf)
```

6.3. Otras funciones gráficas 2-D

Existen otras funciones gráficas bidimensionales orientadas a generar otro tipo de gráficos distintos de los que produce la función ***plot()*** y sus análogas. Algunas de estas funciones son las siguientes (para más información sobre cada una de ellas en particular, utilizar ***help nombre_función***):

bar()	crea diagramas de barras
barh()	diagramas de barras horizontales
bar3()	diagramas de barras con aspecto 3-D
bar3h()	diagramas de barras horizontales con aspecto 3-D
pie()	gráficos con forma de "tarta"
pie3()	gráficos con forma de "tarta" y aspecto 3-D
area()	similar ***plot()***, pero rellenando en ordenadas de 0 a y
stairs()	función análoga a ***bar()*** sin líneas internas
errorbar()	representa sobre una gráfica –mediante barras– valores de errores
compass()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de un origen común
feather()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de orígenes uniformemente espaciados sobre el eje de abscisas
hist()	dibuja histogramas de un vector
rose()	histograma de ángulos (en radianes)
quiver()	dibujo de campos vectoriales como conjunto de vectores

A modo de ejemplo, genérese un vector de valores aleatorios entre 0 y 10, y ejecútense los siguientes comandos:

```
» x=[rand(1,100)*10];
» plot(x)
» bar(x)
» stairs(x)
» hist(x)
» hist(x,20)
» alfa=(rand(1,20)-0.5)*2*pi;
» rose(alfa)
```

6.3.1. FUNCIÓN *FPLOT*

La función ***plot*** vista anteriormente dibuja vectores. Si se quiere dibujar una función, antes de ser pasada a ***plot*** debe ser convertida en un vector de valores. Esto tiene algunos inconvenientes, por ejemplo, el que "a priori" es difícil predecir en que zonas la función varía más rápidamente y habría por ello que reducir el espaciado entre los valores en el eje de abscisas.

La función ***fplot*** admite como argumento un *nombre de función* o un *nombre de fichero* ****.m*** en el cual esté definida una función de usuario. La función puede ser escalar (un único resultado por cada valor de **x**) o vectorial. La forma general de esta función es la siguiente:

```
fplot('funcion', limites, 'cadena', tol)
```

donde: 'funcion' representa el nombre de la función o del fichero ***.m** entre apóstrofos (pasado como cadena de caracteres),

limites es un vector de 2 ó 4 elementos que puede tomar los valores [xmin,xmax] o [xmin,xmax,ymin,ymax],

'cadena' tiene el mismo significado que en ***plot*** y permite controlar el color, los markers y el tipo de línea.

tol es la tolerancia de error relativo. El valor por defecto es 2e-03. El máximo número de valores en **x** es (1/tol)+1

Esta función puede utilizarse también en la forma:

```
[x,y]=fplot('funcion', limites, 'cadena', tol)
```

y en este caso se devuelven los vectores **x** e **y**, pero no se dibuja nada. El gráfico puede obtenerse con un comando posterior por medio de la función ***plot***. Véase un ejemplo de utilización de esta función. Se comienza creando un fichero llamado ***mifunc.m*** en el directorio ***G:\matlab*** que contenga las líneas siguientes:

```
function y = mifunc(x)
y(:,1)=200*sin(x)./x;
y(:,2)=x.^2;
```

y a continuación se ejecuta el comando:

```
» fplot('mifunc(x)', [-20 20], 'g')
```

Obsérvese que la función ***mifunc*** devuelve una matriz con dos columnas, que constituyen las dos gráficas dibujadas. En este caso se ha utilizado para ellas el color verde.

6.3.2. FUNCIÓN *FILL* PARA POLÍGONOS

Ésta es una función especial para dibujar polígonos planos, rellenándolos de un determinado color. La forma general es la siguiente:

```
» fill(x,y,c)
```

que dibuja un polígono definido por los vectores **x** e **y**, rellenándolo con el color especificado por **c**. Si es necesario, el polígono se cierra uniendo el último vértice con el primero. Respecto al color:

- Si **c** es un carácter de color ('r','g','b','c','m','y','w','k'), o un vector de valores [r g b], el polígono se rellena de modo uniforme con el color especificado.
- Si **c** es un vector de la misma dimensión que **x** e **y**, sus elementos se trasforman de acuerdo con un mapa de colores determinado, y el llenado del polígono –no uniforme en este caso– se obtiene interpolando entre los colores de los vértices. Sobre este tema de los colores, se volverá más adelante con un cierto detenimiento.

Este comando puede utilizarse también con matrices:

```
» fill(A,B,C)
```

donde **A** y **B** son matrices del mismo tamaño. En este caso se dibuja un polígono por cada par de columnas de dichas matrices. **C** puede ser un vector fila de colores uniformes para cada polígono, o una matriz del mismo tamaño que las anteriores para obtener colores de relleno por interpolación. Si una de las dos, o **A** o **B**, son un vector en vez de una matriz, se supone que ese vector se repite tantas veces como sea necesario para dibujar tantos polígonos como columnas tiene la matriz. Considérese un ejemplo sencillo de esta función:

```
» x=[1 5 4 2]; y=[1 0 4 3];
» fill(x,y,'r')
» colormap(gray), fill(x,y,[1 0.5 0.8 0.7])
```

6.4. Entrada de puntos con el ratón

Se realiza mediante la función ***ginput***, que permite introducir las coordenadas del punto sobre el que está el cursor, al clicar (o al pulsar una tecla). Algunas formas de utilizar esta función son las siguientes:

[x,y] = ginput	lee un número indefinido de puntos –cada vez que se clica o se pulsa una tecla cualquiera– hasta que se termina pulsando la tecla ***intro***
[x,y] = ginput(n)	lee las coordenadas de **n** puntos
[x,y,bot] = ginput	igual que el anterior, pero devuelve también un vector de enteros **bot** con el código ASCII de la tecla pulsada o el número del botón del ratón (1, 2, ...) con el que se ha clicado

Como ejemplo de utilización de este comando, ejecútense las instrucciones siguientes en la ventana de comandos de MATLAB para introducir un cuadrilátero arbitrario y dibujarlo de dos formas:

```
» clf, [x,y]=ginput(4);
» figure(gcf), plot(x,y,'w'), pause(5), fill(x,y,'r')
```

donde se ha introducido el comando ***pause(5)*** que espera 5 segundos antes de pasar a ejecutar el siguiente comando. Este comando admite como argumento un tiempo con precisión de centésimas de segundo.

6.5. Preparación de películas o "movies"

Para preparar pequeñas películas o movies se pueden utilizar las funciones ***movie***, ***moviein*** y ***getframe***. Una película se compone de varias imágenes, denominadas *frames*. La función ***getframe*** devuelve un vector columna con la información necesaria para reproducir la imagen que se acaba de representar en la figura o ventana gráfica activa, por ejemplo con la función ***plot***. El tamaño de este vector columna depende del tamaño de la ventana, pero no de la complejidad del dibujo. La función ***moviein(n)*** reserva memoria para almacenar **n** frames. La siguiente lista de comandos crearía una película de 17 imágenes o frames, que se almacenarán como las columnas de la matriz **M**:

```
M = moviein(17);
x=[-2*pi:0.1:2*pi]';
for j=1:17
    y=sin(x+j*pi/8);
    plot(x,y);
    M(:,j) = getframe;
end
```

Una vez creada la película se puede representar el número de veces que se desee con el comando ***movie***. Por ejemplo, para representar 10 veces la película anterior, a 15 imágenes por segundo, habría que ejecutar el comando siguiente (los dos últimos parámetros son opcionales):

```
movie(M,10,15)
```

Los comandos ***moviein***, ***getframe*** y ***movie*** tienen posibilidades adicionales para las que puede consultarse el ***Help*** correspondiente. Hay que señalar que en MATLAB no es lo mismo un *movie* que una *animación*. Una *animación* es simplemente una ventana gráfica que va cambiando como consecuencia de los comandos que se van ejecutando. Un *movie* es una animación grabada o almacenada en memoria previamente.

6.6. Impresión de las figuras en impresora láser

Es muy fácil enviar a la impresora o a un fichero una figura producida con MATLAB. La Figura 20 muestra las opciones que ofrece el menú ***File*** relacionadas con la impresión de figuras: es posible establecer los parámetros de la página (***Page Setup***), de la impresora (***Print Setup***), obtener una visión preliminar (***Print Preview***) e imprimir (***Print***). Todos estos comandos se utilizan en la forma habitual de las aplicaciones de ***Windows***.

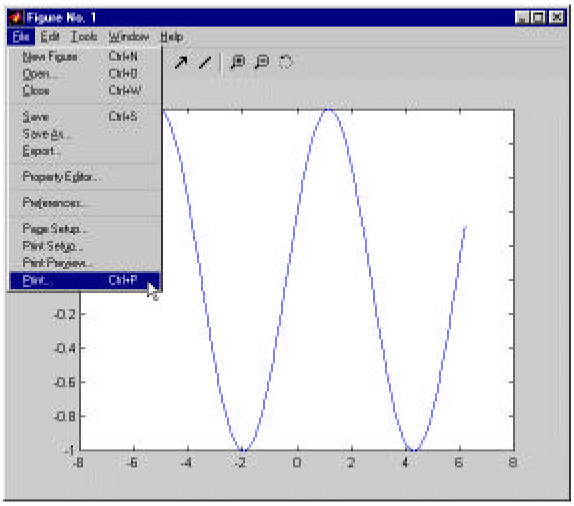


Figura 20. Comandos para imprimir figuras.

Por defecto, MATLAB produce salidas tipo *postcript* (un formato de descripción de páginas propio de impresoras de la gama alta), pero si no hay ninguna impresora *postcript* disponible, MATLAB puede trasformar la salida y convertirla al formato de la impresora disponible en ese momento (por ejemplo al formato *PCL*, propio de las impresoras láser de Hewlett-Packard de la gama media-baja).

La impresión de una figura puede hacerse también desde la línea de comandos. La forma general del comando de impresión es la siguiente (si se omite el nombre del fichero, la figura se envía a la impresora):

```
» print –device –options filename
```

Mediante el ***Help*** se puede obtener más información sobre el comando ***print***.

Es posible también exportar a un fichero una figura de MATLAB, por ejemplo para incluirla luego en un documento de ***Word*** o en una presentación de ***Powerpoint***. Para ello se utiliza el comando ***File/Export*** de la ventana en la que aparece la figura. El cuadro de diálogo que se abre ofrece distintos formatos gráficos para guardar la imagen. Cabe destacar la ausencia del formato ****.gif***, muy utilizado en Internet; sí está presente sin embargo el formato ****.png***, que se considera el sucesor natural del ****.gif***. En todo caso la figura puede exportarse con cualquier formato estándar y luego utilizar por ejemplo ***Paint Shop Pro*** para transformarla.

Si la figura es en color y se envía a una impresora en blanco y negro, se realiza una conversión de colores a tonalidades de gris. En este caso, puede sin embargo ser más adecuado el realizar la figura con un mapa de colores que se ajuste directamente a las distintas tonalidades de gris.

6.7. Las ventanas gráficas de MATLAB

Anteriormente han aparecido en varias ocasiones las ventanas gráficas de MATLAB. Quizás sea el momento de hacer una breve recapitulación sobre sus posibilidades, muy mejoradas en la versión 5.3. La Figura 21 muestra los menús y la barra de herramientas de las ventanas gráficas de MATLAB.

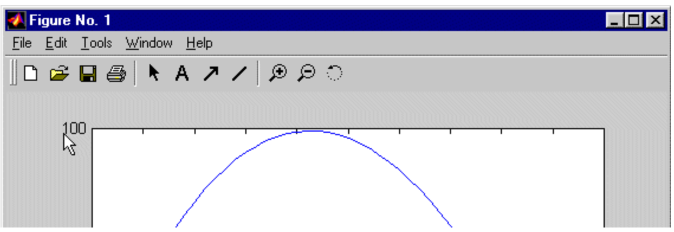


Figura 21. Menús y barras de herramientas de las ventanas gráficas de MATLAB.

En el menú ***File***, además de los comandos referentes a la impresión y a la exportación de figuras, se puede señalar el comando ***Property Editor***, que abre paso al

editor de propiedades que se cometa más adelante en el Apartado 9.2, en la página 108, al hablar de la construcción interactiva de interfaces gráficas de usuario.

El menú ***Edit*** ofrece las opciones estándar de ***Windows***, permitiendo copiar, cortar y pegar los elementos seleccionados de la figura si está activada la opción ***Enable Plot Editing*** ().

El menú ***Tools*** permite mostrar u ocultar la barra de herramientas (***Show Toolbar***), activar la opción de edición de la figura (***Enable Plot Editing***), abrir cuadros de diálogo para editar las propiedades de los ejes (***Axes Properties***), de las líneas (***Line Properties***) y del texto (***Text Properties***), activar o desactivar rótulos (***Show Legend***) y añadir ejes, líneas, flechas y texto (***Add***).

Los botones de la barra de herramientas responden a algunas de estas mismas funciones.

7. GRÁFICOS TRIDIMENSIONALES

Quizás sea ésta una de las características de MATLAB que más admiración despierta entre los usuarios no técnicos (cualquier alumno de ingeniería sabe que hay ciertas operaciones algebraicas –como la descomposición de valor singular, sin ir más lejos– que tienen dificultades muy superiores, aunque "luzcan" menos).

7.1. Tipos de funciones gráficas tridimensionales

MATLAB tiene posibilidades de realizar varios tipos de gráficos 3D. Para darse una idea de ello, lo mejor es verlo en la pantalla cuanto antes, aunque haya que dejar las explicaciones detalladas para un poco más adelante.

La primera forma de gráfico 3D es la función ***plot3***, que es el análogo tridimensional de la función ***plot***. Esta función dibuja puntos cuyas coordenadas están contenidas en 3 vectores, bien uniéndolos mediante una línea continua (defecto), bien mediante *markers*. Asegúrese de que no hay ninguna ventana gráfica abierta y ejecute el siguiente comando que dibuja una línea espiral:

```
» fi=[0:pi/20:6*pi]; plot3(cos(fi),sin(fi),fi,'g')
```

Ahora se verá cómo se representa una función de dos variables. Para ello se va a definir una función de este tipo en un fichero llamado ***test3d.m***. La fórmula será la siguiente:

$$z = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

El fichero ***test3d.m*** debe contener las líneas siguientes:

```
function z=test3d(x,y)
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Ahora, ejecútese la siguiente lista de comandos (directamente, o mejor creando un fichero ***test3dFC.m*** que los contenga):

```
» x=[-3:0.4:3]; y=x;
» close
» subplot(2,2,1)
» figure(gcf),fi=[0:pi/20:6*pi];
» plot3(cos(fi),sin(fi),fi,'r')
» [X,Y]=meshgrid(x,y);
» Z=test3d(X,Y);
» subplot(2,2,2)
» figure(gcf), mesh(Z)
» subplot(2,2,3)
» figure(gcf), surf(Z)
» subplot(2,2,4)
» figure(gcf), contour3(Z,16)
```

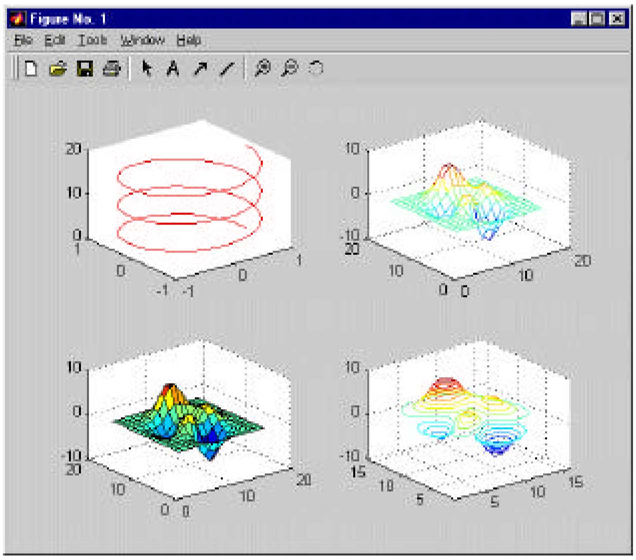


Figura 22. Gráficos 3D realizados con MATLAB.

En la figura resultante (Figura 22) aparece una buena muestra de algunas de las posibilidades gráficas tridimensionales de MATLAB. En las próximas secciones se encontrará la explicación de qué se ha hecho y cómo se ha hecho.

7.1.1. DIBUJO DE LÍNEAS: FUNCIÓN *PLOT3*

La función ***plot3*** es análoga a su homóloga bidimensional ***plot***. Su forma más sencilla es la siguiente:

```
» plot3(x,y,z)
```

que dibuja una línea que une los puntos (x(1), y(1), z(1)), (x(2), y(2), z(2)), etc. y la proyecta sobre un plano para poderla representar en la pantalla. Al igual que en el caso plano, se puede incluir una cadena de 1, 2 ó 3 caracteres para determinar el color, los markers, y el tipo de línea:

```
» plot3(x,y,z,s)
```

También se pueden utilizar tres matrices **X**, **Y** y **Z** del mismo tamaño:

```
» plot3(X,Y,Z)
```

en cuyo caso se dibujan tantas líneas como columnas tienen estas 3 matrices, cada una de las cuales está definida por las 3 columnas homólogas de dichas matrices.

A continuación se va a realizar un ejemplo sencillo consistente en dibujar un *cubo*. Para ello se creará un fichero llamado ***cubo.m*** que contenga las aristas correspondientes, definidas mediante los vértices del cubo como una línea poligonal continua (obsérvese que algunas aristas se dibujan dos veces). El fichero ***cubo.m*** define una matriz **A** cuyas columnas son las coordenadas de los vértices, y cuyas filas son las coordenadas **x**, **y** y **z** de los mismos:

```
A=[0 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0
   0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1
   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0];
```

Ahora basta ejecutar los comandos siguientes (el trasponer los vectores en este caso es opcional):

```
» cubo;
» plot3(A(1,:)',A(2,:)',A(3,:)')
```

7.1.2. DIBUJO DE MALLADOS: FUNCIONES *MESHGRID*, *MESH* Y *SURF*

Ahora se verá con detalle cómo se puede dibujar una función de dos variables (z=f(x,y)) sobre un dominio rectangular. Se verá que también se pueden dibujar los elementos de una matriz como función de los dos índices.

Sean **x** e **y** dos vectores que contienen las coordenadas en una y otra dirección de la retícula (*grid*) sobre la que se va a dibujar la función. Después hay que crear dos matrices **X** (cuyas filas son copias de **x**) e **Y** (cuyas columnas son copias de **y**). Estas matrices se crean con la función ***meshgrid***. Estas matrices representan respectivamente las coordenadas *x* e *y* de todos los puntos de la retícula. La matriz de valores **Z** se calcula a partir de las matrices de coordenadas **X** e **Y**. Finalmente hay que dibujar esta matriz **Z** con la función ***mesh***, cuyos elementos son función elemento a elemento de los elementos de **X** e **Y**. Véase como ejemplo el dibujo de la función ***sen(r)/r*** (siendo $r=sqrt(x^2+y^2)$; para evitar dividir por 0 se suma al denominador el número pequeño ***eps***). Para distinguirla de la función ***test3d*** anterior se utilizará **u** y **v** en lugar de **x** e **y**. Créese un fichero llamado ***sombrero.m*** que contenga las siguientes líneas:

```
close all
u=-8:0.5:8; v=u;
[U,V]=meshgrid(u,v);
R=sqrt(U.^2+V.^2)+eps;
W=sin(R)./R;
mesh(W)
```

Ejecutando este fichero se obtiene el gráfico mostrado en la Figura 23.

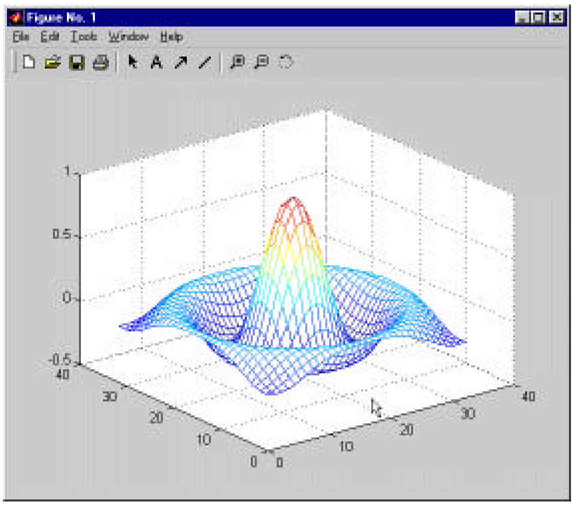


Figura 23. Figura 3D de la función "sombrero".

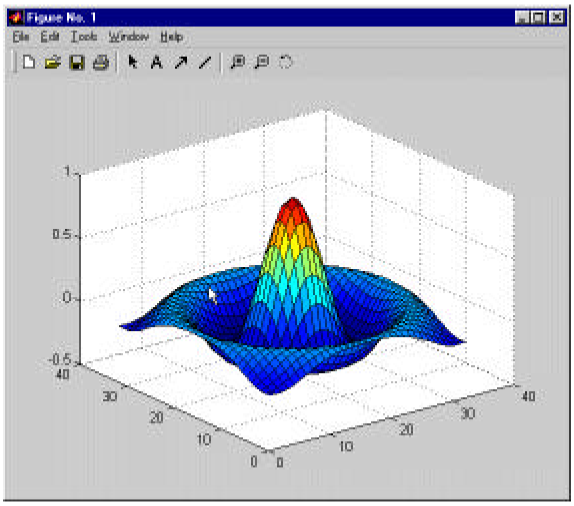


Figura 24. Función "sombrero" con facetas.

Se habrá podido comprobar que la función ***mesh*** dibuja *en perspectiva* una función en base a una retícula de líneas de colores, rodeando cuadriláteros del color de fondo, con eliminación de líneas ocultas. Más adelante se verá cómo controlar estos colores que aparecen. Baste decir por ahora que el color depende del valor **z** de la función. Ejecútese ahora el comando:

```
» surf(W)
```

y obsérvese la diferencia en la Figura 24. En vez de líneas aparece ahora una superficie *faceteada*, también con eliminación de líneas ocultas. El color de las facetas depende también del valor de la función.

Como un segundo ejemplo, se va a volver a dibujar la función ***picos*** (la correspondiente al fichero ***test3d.m*** visto previamente). Créese ahora el fichero ***picos.m*** con las siguientes sentencias:

```
x=[-3:0.2:3];
y=x;
[X,Y]=meshgrid(x,y);
Z=test3d(X,Y);
figure(gcf), mesh(Z), pause(5), surf(Z)
```

Es necesario poner la instrucción ***pause*** –que espera 5 segundos– para que se puedan ver las dos formas de representar la función **Z** (si no, sólo se vería la segunda). Una vez creado este fichero, tecléese ***picos*** en la línea de comandos y obsérvese el resultado. Más adelante se verá también cómo controlar el punto de vista en estos gráficos en perspectiva.

7.1.3. DIBUJO DE LÍNEAS DE CONTORNO: FUNCIONES *CONTOUR* Y *CONTOUR3*

Una forma distinta de representar funciones tridimensionales es por medio de isolíneas o curvas de nivel. A continuación se verá cómo se puede utilizar estas representaciones con las matrices de datos **Z** y **W** que se han calculado previamente:

```
» contour(Z,20)
» contour3(Z,20)
» contour(W,20)
» contour3(W,20)
```

donde "20" representa el número de líneas de nivel. Si no se pone se utiliza un número por defecto. Otras posibles formas de estas funciones son las siguientes:

contour(Z, val)	siendo **val** un vector de valores para las isolíneas a dibujar
contour(u,v,W,20)	se utilizan **u** y **v** para dar valores a los ejes de coordenadas

contour(Z,20,'r--')	se puede especificar el tipo de línea como en la función ***plot***
contourf(Z, val)	análoga a ***contour()***, pero rellenando el espacio entre líneas

7.2. Utilización del color en gráficos 3-D

En los dibujos realizados hasta ahora, se ha visto que el resultado adoptaba determinados colores, pero todavía no se ha explicado de dónde han salido. Ahora se verá qué sistema utiliza MATLAB para determinar los colores.

7.2.1. MAPAS DE COLORES

Un ***mapa de colores*** se define como una matriz de tres columnas, cada una de las cuales contiene un valor entre 0 y 1, que representa la intensidad de uno de los colores fundamentales: R(red o rojo), G (green o verde) y B (blue o azul).

La longitud por defecto de los mapas de colores de MATLAB es 64, es decir, cada mapa de color contiene 64 colores.

Algunos mapas de colores están predefinidos en MATLAB. Buscando ***colormap*** en ***Help Desk*** se obtiene –entre otra información– la lista de los siguientes mapas de colores:

autumn	varies smoothly from red, through orange, to yellow.
bone	is a grayscale colormap with a higher value for the blue component.
colorcube	contains as many regularly spaced colors in RGB colorspace as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
cool	consists of colors that are shades of cyan and magenta.
copper	varies smoothly from black to bright copper.
flag	consists of the colors red, white, blue, and black.
gray	returns a linear grayscale colormap.
hot	varies smoothly from black, through shades of red, orange, and yellow, to white.
hsv	varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red.
jet	ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the hsv colormap.
lines	colormap of colors specified by the Axes ColorOrder property and a shade of gray.
pink	contains pastel shades of pink.
prism	repeats the six colors red, orange, yellow, green, blue, and violet.
spring	consists of colors that are shades of magenta and yellow.
summer	consists of colors that are shades of green and yellow.
white	is an all white monochrome colormap.
winter	consists of colors that are shades of blue and green.

El colormap por defecto es ***jet***. Para visualizar estos mapas de colores se pueden utilizar los comandos:

```
» colormap(hot)
» pcolor([1:65;1:65]')
```

donde la función ***pcolor*** permite visualizar por medio de colores la magnitud de los elementos de una matriz (en realidad representa colores de "celdas", para lo que necesita que la matriz tenga una fila y columna más de las necesarias; ésa es la razón de que en el ejemplo anterior a la función ***pcolor*** se le pasen 65 filas y 2 columnas).

Si se desea imprimir una figura en una impresora láser en blanco y negro, puede utilizarse el mapa de color ***gray***. En el siguiente apartado se explica con más detalle el dibujo en "pseudocolor" (*pcolor*, abreviadamente).

El comando ***colormap*** actúa sobre la figura activa, cambiando sus colores. Si no hay ninguna figura activa, sustituye al mapa de color anterior para las siguientes figuras que se vayan a dibujar.

7.2.2. IMÁGENES Y GRÁFICOS EN *PSEUDOCOLOR*. FUNCIÓN *CAXIS*

Cuando se desea dibujar una figura con un determinado mapa de colores se establece una correspondencia (o un *mapping*) entre los valores de la función y los colores del mapa de colores. Esto hace que los valores pequeños se dibujen con los colores *bajos* del mapa, mientras que los valores grandes se dibujan con los colores *altos*.

La función ***pcolor*** es -en cierta forma- equivalente a la función ***surf*** con el punto de vista situado perpendicularmente al dibujo. Un ejemplo interesante de uso de la función ***pcolor*** es el siguiente: se genera una matriz **A** de tamaño 100x100 con valores aleatorios entre 0 y 1. La función ***pcolor(A)*** dibuja en color los elementos de la matriz **A**, mientras que la función ***pcolor(inv(A))*** dibuja los colores correspondientes a los elementos de la matriz inversa. Se puede observar que los colores de la matriz inversa son mucho más uniformes que los de la matriz original. Los comandos son los siguientes:

```
» A=rand(100,100); colormap(hot); pcolor(A); pause(5), pcolor(inv(A));
```

donde el comando ***pause(5)*** simplemente introduce un pausa de 5 seg en la ejecución. Al ejecutar todos los comandos en la misma línea es necesario poner ***pause*** pues si no dibuja directamente la inversa sin pasar por la matriz inicial.

Si todavía se conservan las matrices **Z** y **W** que se han definido previamente, se pueden hacer algunas pruebas cambiando el mapa de colores.

La función ***caxis*** permite ajustar manualmente la escala de colores. Su forma general es:

```
caxis([cmin, cmax])
```

donde ***cmin*** y ***cmax*** son los valores numéricos a los que se desea ajustar el mínimo y el máximo valor de la escala de colores.

7.2.3. DIBUJO DE SUPERFICIES FACETEADAS

La función ***surf*** tiene diversas posibilidades referentes a la forma en que son representadas las facetas o polígonos coloreados. Las tres posibilidades son las siguientes:

shading flat	determina sombreado con color constante para cada polígono. Este sombreado se llama plano o *flat*.
shading interp	establece que el sombreado se calculará por interpolación de colores entre los vértices de cada faceta. Se llama también sombreado de Gouraud
shading faceted	consiste en sombreado constante con líneas negras superpuestas. Esta es la opción por defecto

Edite el fichero ***picos.m*** de forma que aparezcan menos facetas y más grandes. Se puede probar con ese fichero, eliminando la función ***mesh***, los distintos tipos de sombreado o *shading* que se acaban de citar. Para obtener el efecto deseado, basta poner la sentencia ***shading*** a continuación de la sentencia ***surf***.

7.2.4. OTRAS FORMAS DE LAS FUNCIONES *MESH* Y *SURF*

Por defecto, las funciones ***mesh*** y ***surf*** atribuyen color a los bordes y facetas en función de los valores de la función, es decir en función de los valores de la matriz **Z**. Ésta no es sin embargo la única posibilidad. En las siguientes funciones, las dos matrices argumento **Z** y **C** tienen el mismo tamaño:

```
mesh(Z,C)
surf(Z,C)
```

En las figuras resultantes, mientras se dibujan los valores de **Z**, los colores se obtienen de **C**. Un caso típico es aquél en el que se quiere que los colores dependan de la curvatura de la superficie (y no de su valor). MATLAB dispone de la función ***del2***, que aproxima la curvatura por diferencias finitas con el promedio de los 4 elementos contiguos, resultando así una matriz proporcional a la curvatura. Obsérvese el efecto de esta forma de la función ***surf*** en el siguiente ejemplo (si todavía se tiene la matriz **Z** formada a partir de ***test3d***, utilícese. Si no se conserva, vuélvase a calcular):

```
» C=del2(Z);
» close, surf(Z,C)
```

7.2.5. FORMAS PARAMÉTRICAS DE LAS FUNCIONES *MESH*, *SURF* Y *PCOLOR*

Existen unas formas más generales de las funciones ***mesh***, ***surf*** y ***pcolor***. Son las siguientes (se presentan principalmente con la funciones ***mesh*** y ***surf***). La función:

```
mesh(x,y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (x(j), y(i), Z(i,j)) y como color C(i,j). Obsérvese que **x** varía con el índice de columnas e **y** con el de filas. Análogamente, la función:

```
mesh(X,Y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (X(i,j), Y(i,j), Z(i,j)) y como color C(i,j). Las cuatro matrices deben ser del mismo tamaño. Si todavía están disponibles las matrices calculadas con el fichero ***picos.m***, ejecútese el siguiente comando y obsérvese que se obtiene el mismo resultado que anteriormente:

```
» close, surf(X,Y,Z), pause(5), mesh(X,Y,Z)
```

¿Cuál es la ventaja de estas nuevas formas de las funciones ya conocidas? La principal es que admiten más variedad en la forma de representar la cuadrícula en el plano (x-y). La primera forma admite vectores **x** e **y** con puntos desigualmente espaciados, y la segunda admite conjuntos de puntos muy generales, incluso los provenientes de coordenadas *cilíndricas* y *esféricas*.

7.2.6. OTRAS FUNCIONES GRÁFICAS 3D

Las siguientes funciones se derivan directamente de las anteriores, pero añaden algún pequeño detalle y/o funcionalidad:

surfc	combinación de ***surf***, y ***contour*** en z=0
meshz	***mesh*** con plano de referencia en el valor mínimo y una especie de “cortina” en los bordes del dominio de la función
surfl	para controlar la iluminación determinando la posición e intensidad de un foco de luz.
light	crea un foco de luz en los ejes actuales capaz de actuar sobre superficies 3-D. Se le deben pasar como argumentos el color, el estilo (luz local o en el infinito) y la posición. Son muy importantes las propiedades de los objetos iluminados ***patch*** y ***surface*** (*AmbientStrength*, *DiffuseStrength*, *SpecularColorReflectance*, *SpecularExponent*, *SpecularStrength*, *VertexNormals*, *EdgeLighting*, y *FaceLighting*), y de los ejes (*AmbientLightColor*).

Se pueden probar estas funciones con los datos de que se dispone. Utilícese el ***help*** para ello.

7.2.7. ELEMENTOS GENERALES: EJES, PUNTOS DE VISTA, LÍNEAS OCULTAS, ...

Las funciones ***surf*** y ***mesh*** dibujan funciones tridimensionales en perspectiva. La localización del punto de vista o dirección de observación se puede hacer mediante la función ***view***, que tiene la siguiente forma:

```
view(azimut, elev)
```

donde ***azimut*** es el ángulo de rotación de un plano horizontal, medido sobre el eje **z** a partir del eje **x** en sentido antihorario, y ***elev*** es el ángulo de elevación respecto al plano (x-y). Ambos ángulos se miden ***en grados***, y pueden tomar valores positivos y negativos (sus valores por defecto son -37.5 y 30). También se puede definir la dirección del punto de vista mediante las tres coordenadas cartesianas de un vector (sólo se tiene en cuenta la dirección):

```
view([xd,yd,zd])
```

En los gráficos tridimensionales existen funciones para controlar los ejes, por ejemplo:

```
axis([xmin,xmax,ymin,ymax,zmin,zmax])
```

También se pueden utilizar las funciones siguientes: ***xlabel***, ***ylabel***, ***zlabel***, ***axis('auto')***, ***axis(axis)***, etc.

Las funciones ***mesh*** y ***surf*** disponen de un algoritmo de *eliminación de líneas ocultas* (los polígonos o facetas, no dejan ver las líneas que están detrás). El comando ***hidden*** activa y desactiva la eliminación de líneas ocultas.

En el dibujo de funciones tridimensionales, a veces también son útiles los *NaNs*. Cuando una parte de los elementos de la matriz de valores **Z** son *NaNs*, esa parte de la superficie no se dibuja, permitiendo ver el resto de la superficie.

8. FUNDAMENTOS DE LAS INTERFACES GRÁFICAS CON MATLAB

MATLAB permite desarrollar de manera simple un conjunto de pantallas con botones, menús, ventanas, etc., que permiten utilizar de manera muy simple programas realizados en el entorno ***Windows***. Este conjunto de herramientas se denomina ***interface de usuario***. Las posibilidades que ofrece MATLAB 5.2 han mejorado mucho respecto a versiones anteriores, aunque no son muy amplias en comparación con otras aplicaciones de ***Windows*** como ***Visual Basic***.

Para poder hacer programas que utilicen las capacidades gráficas avanzadas de MATLAB hay que conocer algunos conceptos que se explican en los apartados siguientes. Aunque MATLAB dispone ahora de la herramienta GUIDE, que permite generar interfaces de usuario de una forma muy cómoda y sencilla, es conveniente conocer los fundamentos de lo que se está haciendo, e incluso ser capaz de hacerlo programando si ayudas.

8.1. Estructura de los gráficos de MATLAB

Los gráficos de MATLAB tienen una estructura jerárquica formada por ***objetos*** de distintos ***tipos***. Esta jerarquía tiene forma de *árbol*, con el aspecto mostrado en la Figura 25.

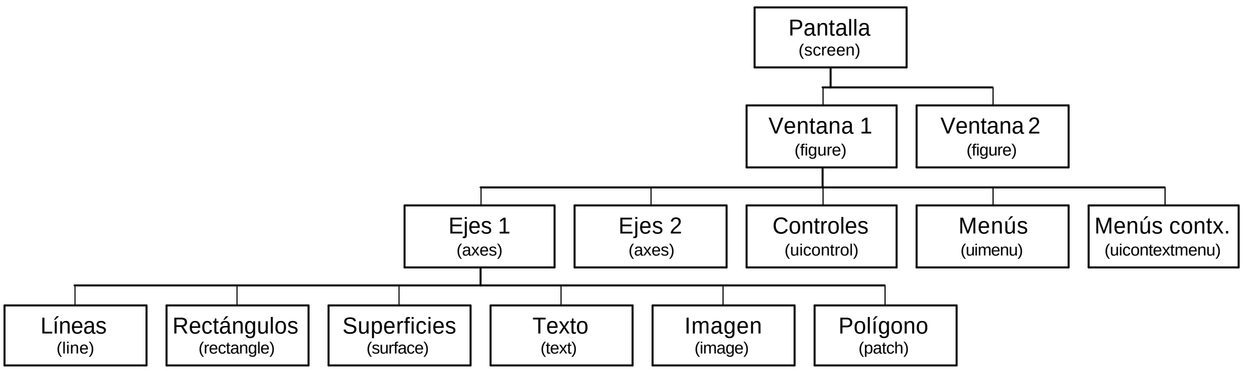


Figura 25. Jerarquía gráfica de MATLAB.

8.1.1. OBJETOS GRÁFICOS DE MATLAB

Según se muestra en la Figura 25, el objeto más general es la ***pantalla*** (***screen***). Este objeto es la raíz de todos los demás y sólo puede haber un objeto pantalla. Una pantalla puede contener una o más ***ventanas*** (***figures***). A su vez cada una de las ventanas puede tener uno o más ***ejes*** de coordenadas (***axes***) en los que representar otros objetos de más bajo nivel. Una ventana puede tener también ***controles*** (***uicontrol***) tales como *botones*, *barras de desplazamiento, botones de selección o de opción*, etc.) y ***menús*** (***uimenu***). Finalmente, los ejes pueden contener los seis tipos de elementos gráficos que permite MATLAB: ***líneas*** (***line***), ***rectángulos*** (***rectangle***), ***polígonos*** (***patches***), ***superficies*** (***surface***), ***imágenes bitmap*** (***image***) y ***texto*** (***text***).

La jerarquía de objetos mostrada en la Figura 25 indica que en MATLAB hay ***objetos padres*** e ***hijos***. Por ejemplo, todos los objetos ***ventana*** son hijos de ***pantalla***, y cada ***ventana*** es padre de los objetos ***ejes***, ***controles*** o ***menús*** que están por debajo. A su vez los elementos gráficos (líneas, polígonos, etc.) son hijos de un objeto ***ejes***, y no tienen otros objetos que sean sus hijos.

Cuando se borra un objeto de MATLAB automáticamente se borran todos los objetos que son sus descendientes. Por ejemplo, al borrar unos ejes, se borran todas las líneas y polígonos que son hijos suyos.

8.1.2. IDENTIFICADORES (*HANDLES*)

Cada uno de los objetos de MATLAB tiene un ***identificador único*** (***handle***). En este escrito a los identificadores se les llamará ***handle*** o ***id***, indistintamente. Algunos gráficos tienen muchos objetos, en cuyo caso tienen múltiples *handles*. El ***objeto raíz*** (pantalla) es siempre único y su identificador es el ***cero***. El identificador de las ventanas es un entero, que aparece en la barra de nombre de dicha ventana. Los identificadores de otros elementos gráficos son números *float*, que pueden ser obtenidos como valor de retorno y almacenados en variables de MATLAB.

MATLAB puede tener varias ventanas abiertas, pero siempre hay una y sólo una que es la ***ventana activa***. A su vez una ventana puede tener varios *ejes* (*axes*), pero sólo unos son los ***ejes activos***. MATLAB dibuja en los ejes activos de la ventana activa. Los identificadores de la ventana activa, de los ejes activos y del objeto activo se pueden obtener respectivamente con los comandos ***gcf*** (*get current figure*), ***gca*** (*get current axes*) y ***gco*** (*get current object*):

gcf devuelve un entero, que es el ***handle*** de la ventana activa

gca devuelve el ***handle*** de los ejes activos

gco devuelve el ***handle*** del objeto activo

Los objetos se pueden borrar con el comando ***delete***:

delete(handle) borra el objeto correspondiente y todos sus hijos

MATLAB dispone de funciones gráficas de alto y bajo nivel. Son funciones de alto nivel las funciones ***plot***, ***plot3***, ***mesh***, ***surf***, ***fill***, ***fill3***, etc., utilizadas previamente. Cada una de estas funciones llama a una o más funciones de bajo nivel. Las funciones de bajo nivel crean cada uno de los 9 tipos de objetos disponibles y de ordinario tienen el nombre inglés del objeto correspondiente: ***figure***, ***axes***, ***uicontrol***, ***uimenu***, ***line***, ***rectangle***, ***patch***, ***surface***, ***image*** y ***text***.

Como ejemplo, ejecútense los siguientes comandos, observando la evolución de lo dibujado en la ventana y como MATLAB devuelve el ***id*** de cada objeto como valor de retorno:

```
» fig = figure
» li1 = line([0,5],[0,5])
» li2 = line([0,5],[5,0])
» po1 = patch([1,4,3],[1,1,4],'g')
» delete(po1)
» delete(li1)
```

Estos valores de retorno pueden ser almacenados en variables para un uso posterior. En ocasiones el ***id*** es un vector de valores, como por ejemplo al crear una superficie que consta de líneas y polígonos. Los comandos anteriores han abierto una ventana y dibujado sobre ella dos líneas cruzadas de color amarillo y un triángulo de color verde.

8.2. Propiedades de los objetos

Todos los objetos de MATLAB tienen distintas ***propiedades***. Algunas de éstas son el *tipo*, el *estilo*, el *padre*, los *hijos*, si es *visible* o no, y otras propiedades particulares del objeto concreto de que se trate. Algunas de las propiedades comunes a todos los objetos son: ***children***, ***clipping***, ***parent***, ***type***, ***UserData***, ***Visible*** y ***Tag***. Otras propiedades son propias de un tipo determinado de objeto.

Las propiedades tienen ***valores por defecto***, que se utilizan siempre que el usuario no indique otra cosa. Es posible cambiar las propiedades por defecto, y también devolverles su valor original (llamado ***factory***, por ser el valor por defecto con que salen de fábrica). El usuario puede consultar (***query***) los valores de las propiedades de cualquier objeto. Algunas propiedades pueden ser

modificadas y otras no (son ***read only***). Hay propiedades que pueden tener cualquier valor y otras que sólo pueden tener un conjunto limitado de valores (por ejemplo, ***on*** y ***off***).

8.2.1. FUNCIONES *SET()* Y *GET()*

MATLAB dispone de las funciones ***set*** y ***get*** para consultar y cambiar el valor de las propiedades de un objeto. Las funciones ***set(id)*** lista en pantalla todas las propiedades del objeto al que corresponde el *handle* (sólo los *nombres*, sin los valores de las propiedades). La función ***get(id)*** produce un listado de las propiedades y de sus *valores*. Como ejemplo, siendo ***li1*** el ***id*** de la primera línea dibujada anteriormente, la respuesta a ***set(li1)*** es:

```
» set(li1)
    Color
    EraseMode: [ {normal} | background | xor | none ]
    LineStyle: [ {-} | -- | : | -. | none ]
    LineWidth
    Marker: [ + | o | * | . | x | square | diamond | v | ^ | > | < | pentagram |
           hexagram | {none} ]
    MarkerSize
    MarkerEdgeColor: [ none | {auto} ] -or- a ColorSpec.
    MarkerFaceColor: [ {none} | auto ] -or- a ColorSpec.
    XData
    YData
    ZData

    ButtonDownFcn
    Children
    Clipping: [ {on} | off ]
    CreateFcn
    DeleteFcn
    BusyAction: [ {queue} | cancel ]
    HandleVisibility: [ {on} | callback | off ]
    HitTest: [ {on} | off ]
    Interruptible: [ {on} | off ]
    Parent
    Selected: [ on | off ]
    SelectionHighlight: [ {on} | off ]
    Tag
    UIContextMenu
    UserData
    Visible: [ {on} | off ]
```

que muestra las propiedades del objeto *línea*. Las propiedades que tienen un conjunto finito de valores presentan estos valores entre *corchetes* [ ] separados por barras verticales. La opción por defecto se muestra entre *llaves* { }. En general, el significado de cada propiedad es bastante evidente a partir de su nombre.

El comando ***get(id)*** devuelve las propiedades del objeto junto con sus valores:

```
» get(li1)
    Color = [0 0 1]
    EraseMode = normal
    LineStyle = -
    LineWidth = [0.5]
    Marker = none
    MarkerSize = [6]
    MarkerEdgeColor = auto
    MarkerFaceColor = none
    XData = [0 5]
    YData = [0 5]
    ZData = []

    ButtonDownFcn =
    Children = []
```

```
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [12.0001]
Selected = off
SelectionHighlight = on
Tag =
Type = line
UIContextMenu = []
UserData = []
Visible = on
```

Para conocer el valor de una propiedad particular de un objeto se utiliza la función ***get(id,'propiedad')***.

```
» get(li1,'color')
ans =
     1     1     0
```

Las propiedades de un objeto pueden ser cambiadas o modificadas (salvo que sean *read-only*) con el comando ***set(id,'propiedad','valor')***. Por ejemplo, para cambiar el color de la segunda línea en el ejemplo anterior:

```
» set(li2,'color','r')
```

Es interesante hacer pruebas con los distintos tipos de objetos gráficos que se pueden crear y manipular con MATLAB. Por ejemplo, ejecútense los siguientes comandos observando cómo van cambiando el grosor de las líneas y los colores:

```
» close(gcf)
» fig=figure
» li1=line([0,5],[0,5])
» li2=line([0,5],[5,0],’color’,’w’)
» po1=patch([1,4,3],[1,1,4],'g')
» pause(3)
» set(li1,'LineWidth',2), pause(1);
» set(li2,'LineWidth',2,'color','r'), pause(1);
» set(po1,'LineWidth',2,'EdgeColor','w','FaceColor','b')
```

El comando ***set*** permite cambiar varias propiedades a la vez, poniendo sus nombres entre apóstrofos seguidos de sus valores. Los ejemplos anteriores demuestran que es esencial disponer de los ***id*** si se desea modificar un gráfico o utilizar propiedades distintas de las de defecto.

Es posible también establecer las propiedades en el momento de la creación del objeto, como en el ejemplo siguiente que crea una figura con fondo blanco:

```
» fig = figure('color','w')
```

Se puede utilizar la propiedad ***type*** para saber qué tipo de objeto (*línea*, *polígono*, *texto*, ...) corresponde a un determinado ***id***. Esto es especialmente útil cuando el ***id*** es un vector de valores correspondientes a objetos de distinto tipo.

```
» get(li2,'type')
line
```

8.2.2. PROPIEDADES POR DEFECTO

Anteponiendo la palabra ***Default*** al nombre de un objeto y de una propiedad se puede acceder al valor por defecto de una propiedad, bien para consultar su valor, bien para modificarlo. Por

ejemplo, ***DefaultLineColor*** representa el color por defecto de una *línea*, y ***DefaultFigureColor*** representa el *color de fondo* por defecto de las ventanas. Cambiando un valor por defecto a un determinado nivel de la jerarquía de objetos se cambia ese valor para todos los objetos que están por debajo y que se creen a partir de ese momento. Por ejemplo, el siguiente comando cambia el color de fondo de todas las ventanas (hijas de *screen*) que sean creadas a partir de ese momento:

```
» set(0,'DefaultFigureColor','w')
```

Cuando se crea un objeto se busca el valor por defecto de sus propiedades a su nivel, y si no se encuentra se sube en la jerarquía hasta que se encuentra un valor por defecto, y ese valor es el que se utiliza. Para devolver una propiedad a su valor original se utiliza el valor ***'factory'***, como por ejemplo:

```
» set(id, 'FaceColor', 'factory')
```

De forma análoga, el valor ***'remove'*** para una propiedad elimina un valor introducido previamente. Por ejemplo, para que el fondo de las ventanas deje de ser blanco se debe ejecutar el comando:

```
» set(0,'DefaultFigureColor','remove')
```

8.2.3. FUNCIONES DE UTILIDAD

MATLAB dispone de algunas funciones que permiten modificar las propiedades de algunos objetos de una forma más directa y sencilla que con las funciones ***get*** y ***set***. Algunas de estas funciones, ya mencionadas al hablar de los gráficos 2-D y 3-D, son ***axis***, ***cla***, ***colormap*** y ***grid***.

Para obtener más información sobre estas funciones puede utilizarse el ***Help*** de MATLAB.

8.3. Creación de controles gráficos: Comando *uicontrol*

MATLAB permite desarrollar programas con el aspecto típico de las aplicaciones de ***Windows***. En este apartado se estudiará cómo crear algunos de los controles más utilizados. Para todos los controles, se utilizará la función ***uicontrol***, que permite desarrollar dichos controles. La forma general del comando ***uicontrol*** es la siguiente:

```
id_control = uicontrol(id_parent,...
                       'Propiedad1',valor1,...
                       'Propiedad2',valor2,...
                        ... (otras propiedades)
                       'callback','sentencias')
```

Las propiedades son las opciones del comando, que se explican en el apartado siguiente. Éstas tendrán comillas simples (') a su izquierda y derecha, e irán seguidas de los parámetros necesarios. En caso de que el conjunto de propiedades de un control exceda una línea de código, es posible continuar en la línea siguiente, poniendo tres puntos seguidos (…).

El comando ***uicontrol*** permite definir los controles gráficos de MATLAB descritos en los siguientes apartados. Estos controles reciben las acciones de los usuarios, que se denominan ***eventos*** (por ejemplo, clicar en un botón, cambiar el valor de una barra de desplazamiento, ...). A continuación se explican algunas de las propiedades de ***uicontrol***.

8.3.1. COLOR DEL OBJETO (*BACKGROUNDCOLOR*)

Controla el color del objeto. Por defecto éste suele ser gris claro, aunque puede tomar distintos valores: ***green***, ***red, white***, etc. Éstos irán definidos entre comillas (por ejemplo '***green***') o con un vector de tres elementos que indican las componentes RGB (*Red*, *Green*, *Blue*).

8.3.2. ACCIÓN A EFECTUAR POR EL COMANDO (*CALLBACK*)

Este comando especifica la ***acción*** a efectuar por MATLAB al actuar sobre el control. Se trata de una expresión u orden, almacenada en una cadena de caracteres o en una función, que se ejecutará al activar el control. Esta instrucción es equivalente a ejecutar la función o a realizar ***eval****('expresión')*. Algunos controles tienen distintos tipos de callback según el evento que reciban del usuario.

8.3.3. CONTROL ACTIVADO/DESACTIVADO (*ENABLE)*

Este comando permite desactivar un control, de tal forma que una acción sobre el mismo (por ejemplo, apretar sobre el mismo con el ratón) no produce ningún efecto. Los valores que puede tomar esta variable son ***on*** u ***off***. Cuando un control está desactivado suele mostrar un aspecto especial (una tonalidad más gris, por ejemplo) que indica esta situación.

8.3.4. ALINEAMIENTO HORIZONTAL DEL TITULO (*HORIZONTALALIGNMENT*)

Esta opción determina la posición del título del control en el mismo (propiedad ***String***). Los valores que puede tomar son: ***left***, ***center*** o ***right***. En los botones y en otros controles la opción por defecto es ***center***.

8.3.5. VALOR MÁXIMO (*MAX*)

Esta opción determina el máximo valor que puede tomar la propiedad ***Value*** cuando se utilizan cajas de textos, botones de opción o barras de desplazamiento. En caso de botones tipo ***on/off***, que solamente admiten las dos posiciones de abierto y cerrado, esta variable corresponde al valor del mismo cuando está activado.

8.3.6. VALOR MÍNIMO (*MIN*)

Análogo a la opción anterior para el valor mínimo.

8.3.7. IDENTIFICADOR DEL OBJETO PADRE (*PARENT*)

Esta opción especifica el ***id*** del objeto *padre*. Debe ir siempre en primer lugar.

8.3.8. POSICIÓN DEL OBJETO (*POSITION*)

En esta opción se determina la posición y el tamaño del control dentro del objeto *padre*. Para ello se define un vector de cuatro elementos, cuyos valores siguen el siguiente orden: [*izquierda*, *abajo*, *anchura*, *altura*]. Aquí *izquierda* y *abajo* son la distancia a la esquina inferior izquierda de la figura y *anchura* y *altura* las dimensiones del control. Por defecto se mide en ***pixels***, aunque con la propiedad ***Units*** las unidades se pueden cambiar.

8.3.9. NOMBRE DEL OBJETO (*STRING*)

Esta opción define el nombre que aparecerá en el control. Cuando el control sea una opción de menú desplegable (***popup menu***), los nombres se sitúan en orden dentro del *String*, separados por un carácter barra vertical (|).

8.3.10. TIPO DE CONTROL (*STYLE*)

Esta opción puede tomar los siguientes valores: ***pushbutton, togglebuttons***, ***radiobutton, checkbox, slider, edit, popupmenu, list, frames*** y ***text***, según el tipo de control que se desee (Como se verá en los ejemplos siguientes, pueden usarse nombres abreviados: así, ***pushbutton*** puede abreviarse en ***push***).

8.3.11. UNIDADES (*UNITS*)

Unidades de medida en las que se interpretará el comando ***Position***. Los valores que puede tomar ***Units*** son: ***pixels*** (puntos de pantalla), ***normalized*** (coordenadas de 0 a 1), ***inches*** (pulgadas), ***cent*** (centímetros), ***points*** (equivalente a 1/72 parte de una pulgada). La opción por defecto es ***pixels***.

8.3.12. VALOR (*VALUE*)

Permite utilizar el valor que tiene del control en un momento dado. Por ejemplo, un botón de chequeo (***checkbutton***) puede tener dos tipos de valores, definidos por las variables ***Max*** y ***Min***. Según sea uno u otro el programa ejecutará una acción. La variable ***Value*** permite controlar dicho valor. Esta propiedad es especialmente importante en las barras de desplazamiento (***sliders***), pues son controles especialmente diseñados para que el usuario introduzca valores. También en los controles ***list*** y ***popupmenu***, en donde el valor representa el número de orden que en la lista de opciones ocupa el elemento elegido por el usuario.

8.3.13. VISIBLE (*VISIBLE*)

Puede tomar dos valores: ***on/off***. Indica si el control es visible o no en la ventana. Este comando es similar al ***Enable***, si bien con ***Visible*** en ***off***, además de quedar inhabilitado, el control desaparece de la pantalla.

8.4. Tipos de *uicontrol*

Existen ocho tipos de controles diferentes. La utilización de cada uno vendrá dada en función de sus características y aplicación.

8.4.1. BOTONES (*PUSHBUTTONS*)



Figura 26. Botón (***push button***).

La Figura 26 muestra el aspecto típico de un botón. Al clicar sobre él con el ratón se producirá un evento que lanza una acción que deberá ser ejecutada por MATLAB.

Ejemplo: La siguiente instrucción dibujará en la figura activa de MATLAB un botón como el de la Figura 26, que tiene como nombre ***Start Plot***. Al pulsarlo se ejecutarán las instrucciones contenidas en el fichero ***mifunc.m***:

```
pbstart = uicontrol(gcf,...
   'Style','push',...
   'Position',[10 10 100 25],...
   'String','Start Plot',...
   'CallBack','mifunc');
```

donde es necesario crear un fichero llamado ***mifunc.m,*** que será ejecutado al pulsar sobre el botón. En lugar de este fichero también puede ponerse una cadena de caracteres conteniendo distintos comandos, que será evaluada como si se tratase de un fichero ****.m***. Como ejemplo se puede crear un fichero ***mifunc.m*** que contenga sólo la sentencia:

```
disp('Ha pulsado en Start Plot')
```

8.4.2. BOTONES DE SELECCIÓN (*CHECK BOXES*)

Los botones de selección permiten al usuario seleccionar entre dos opciones (Figura 27). Los botones de selección actúan como interruptores, indicando un estado ***on*** (si el botón está activado) u ***off*** (si el botón está desactivado). El valor de ambos estados viene definido por las opciones ***Max*** y ***Min***, respectivamente. Los botones de selección deben ser independientes unos de otros.

Ejemplo: El siguiente conjunto de instrucciones crea una caja con dos opciones, ambas permiten el control de los ejes, de manera independiente, dentro de la función ***plot***:

```
% Definir un texto fijo como título para los botones de selección
txt_axes = uicontrol(gcf,...
   'Style','text',...
   'Units','normalized','Position',[0.4   0.55   0.25
0.1],...
   'String','Set Axes Properties');

% Definir la checkbox para la propiedad Box de los
ejes
cb_box = uicontrol(gcf,...
   'Style','checkbox',...
   'Units','normalized','Position',[0.4  0.475  0.25
0.1],...
   'String','Box=on',...
   'CallBack',['set(gca,''Box'',''off''),',...
      'if get(cb_box,''value'')==1,',...
         'set(gca,''Box'',''on''),',...
      'end']);
```

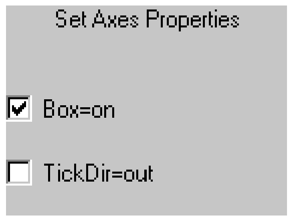


Figura 27. Botones de selección (***Check Boxes***).

```
% Definir la checkbox para la propiedad TickDir de los ejes
cb_tdir = uicontrol(gcf,...
   'Style','checkbox',...
   'Units','normalized','Position',[0.4 0.4 0.25 0.1],...
   'String','TickDir=out',...
   'CallBack',['set(gca,''TickDir'',''in''),',...
      'if get(cb_tdir,''value'')==1,',...
         'set(gca,''TickDir'',''out''),',...
      'end']);
```

8.4.3. BOTONES DE OPCIÓN (*RADIO BUTTONS*)

Al igual que los botones de selección, los botones de opción permiten al usuario seleccionar entre varias posibilidades. La diferencia fundamental reside en que en los botones de opción, las opciones son excluyentes, es decir, no puede haber más de uno activado, mientras que el control anterior permite tener una o más cajas activadas.



Figura 28. Botones de opción (***Radio Buttons***).

Ejemplo: El siguiente ejemplo corresponde a los controles de la Figura 28. Estos botones permiten cambiar de dirección los indicadores de los ejes: ***In*** para orientarlos hacia dentro de la figura, o ***Out*** para que se sitúen en el exterior de la gráfica.

```
% Definir el texto de título para este grupo de controles
txt_tdir = uicontrol(gcf,...
   'Style','text', 'String','Seleccionar posición marcas',...
   'Position',[200 200 150 20]);
```

```
% Definir la propiedad TickDir In con radiobutton (defecto)
td_in = uicontrol(gcf,...
   'Style','radio', 'String','In',...
   'Position',[200 175 150 25],...
   'Value',1,...
   'CallBack',[...
      'set(td_in,''Value'',1),'...
      'set(td_out,''Value'',0),'...
      'set(gca,''TickDir'',''in''),']);

% Definir la propiedad TickDir Out con radiobutton
td_out = uicontrol(gcf,...
   'Style','radio', 'String','Out',...
   'Position',[200 150 150 25],...
   'CallBack',[...
      'set(td_out,''Value'',1),'...
      'set(td_in,''Value'',0),'...
      'set(gca,''TickDir'',''out''),']);
```

8.4.4. BARRAS DE DESPLAZAMIENTO (*SCROLLING BARS O SLIDERS*)

Las barras de desplazamiento (ver Figura 29) permiten al usuario introducir un valor entre un rango de valores. El usuario puede cambiar el valor clicando sobre la barra, clicando en las flechas laterales o bien arrastrando el elemento central con el ratón.

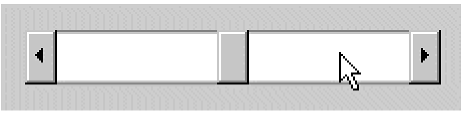


Figura 29. Barra de desplazamiento (***slider***).

Ejemplo: El siguiente ejemplo muestra como se utilizan las barras de desplazamiento para mover un sistema de referencia espacial:

```
% Obtener un id de la ventana activa y borrar su contenido
fig = gcf; clf

% Los callbacks definen la propiedad View de los ejes
%    a partir de os valores de las barras de desplaz. (Value property)
%    y escriben sus valores en los controles text
%
% Definir la barra para el ángulo de azimut
sli_azm = uicontrol(fig,'Style','slider','Position',[50 50 120 20],...
    'Min',-90,'Max',90,'Value',30,...
    'CallBack',[...
    'set(azm_cur,''String'','...
        'num2str(get(sli_azm,''Val''))),',...
    'set(gca,''View'',',...
    '[get(sli_azm,''Val''),get(sli_elv,''Val'')])']);

% Definir la barra para el ángulo de elevación
sli_elv = uicontrol(fig,...
   'Style','slider',...
   'Position',[240 50 120 20],...
   'Min',-90,'Max',90,'Value',30,...
   'CallBack',[...
   'set(elv_cur,''String'',',...
        'num2str(get(sli_elv,''Val''))),',...
   'set(gca,''View'',',...
   '[get(sli_azm,''Val''),get(sli_elv,''Val'')])']);
```

```
% Definir los controles de texto para los valores mínimos
azm_min = uicontrol(fig,...
   'Style','text',...
   'Pos',[20 50 30 20],...
   'String',num2str(get(sli_azm,'Min')));
elv_min = uicontrol(fig,...
   'Style','text',...
   'Pos',[210 50 30 20],...
   'String',num2str(get(sli_elv,'Min')));

% Definir los controles de texto para los valores máximos
azm_max = uicontrol(fig,...
   'Style','text',...
   'Pos',[170 50 30 20],...
   'String',num2str(get(sli_azm,'Max')),...
   'Horiz','right');
elv_max = uicontrol(fig,...
   'Style','text',...
   'Pos',[360 50 30 20],...
   'String',num2str(get(sli_elv,'Max')),...
   'Horiz','right');

% Definir las etiquetas de las barras
azm_label = uicontrol(fig,...
   'Style','text',...
   'Pos',[50 80 65 20],...
   'String','Azimuth');
elv_label = uicontrol(fig,...
   'Style','text',...
   'Pos',[240 80 65 20],...
   'String','Elevación');

% Definir los controles de texto para los valores actuales
% Los valores son inicializados aquí y son modificados por los callbacks
%    de las barras cuando el usuario cambia sus valores
azm_cur = uicontrol(fig,...
   'Style','text',...
   'Pos',[120 80 50 20],...
   'String',num2str(get(sli_azm,'Value')));
elv_cur = uicontrol(fig,...
   'Style','text',...
   'Pos',[310 80 50 20],...
   'String',num2str(get(sli_elv,'Value')));
```

8.4.5. CAJAS DE SELECCIÓN DESPLEGABLES (*POP-UP MENUS*)

Las cajas de selección desplegables permiten elegir una opción entre varias mostradas en una lista. Eligiendo una de las opciones, MATLAB realizará la opción elegida. Según la Figura 30, el menú se despliega pulsando sobre la flecha de la derecha. La opción sobre la que pase el ratón (en este caso ***red***) aparecerá de otro color.

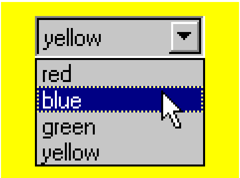


Figura 30. Menú desplegable (***Pop-up menu***).

Ejemplo: El siguiente código abre una ventana con un determinado tamaño y posición y permite cambiar su color de fondo con un menú pop-up (lista desplegable, que aparece desde el primer momento en la pantalla):

```
close all;
pantalla = get(0,'ScreenSize'); xw=pantalla(3); yw=pantalla(4);
fig=figure('Position',[xw/4 yw/4 xw/2 yw/2])
```

```
% Definir el menú pop-up
popcol = uicontrol(fig,...
   'Style','popup',...
   'String','red|blue|green|yellow',...
   'Position',[xw/16 yw/16 xw/12 yw/32],...
   'CallBack',['cb_col = [''R'',''B'',''G'',''Y''];',...
   'set(gcf,''Color'',cb_col(get(popcol,''Value'')))']);
```

8.4.6. CAJAS DE TEXTO (*STATIC TEXTBOXES*)

Son controles especiales, ya que no permite realizar ninguna operación con el ratón. Permiten escribir un texto en la pantalla. Una aplicación de este ***uicontrol*** aparece en la variable ***txt_tdir*** del ejemplo de los ***radiobuttons***. En este caso particular se el texto indica la función de los botones de opción.

8.4.7. CAJAS DE TEXTO EDITABLES (*EDITABLE TEXTBOXES*)

Las cajas de texto se utilizan para introducir y modificar cadenas de caracteres (ver Figura 31). Puede tener una o más líneas, y la llamada a la opción de ejecución ***Callback*** será efectiva una vez que se pulsen las teclas ***Control-Return*** o se clique con el ratón fuera del control.

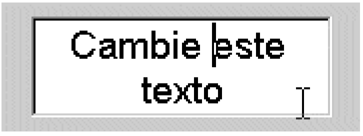


Figura 31. Texto editable (***text***).

Ejemplo: El siguiente ejemplo escribe un texto en la ventana de MATLAB que el usuario puede modificar.

```
fig=gcf; clf;
edmulti = uicontrol(fig,...
   'Style','edit',...
   'BackgroundColor','white',...
   'FontSize',14,'FontName','Arial',...
   'String','Cambie este texto',...
   'Position',[40 200 150 50],...
   'Max',2,...
   'CallBack','get(edmulti,''String'')'...
   );
```

8.4.8. MARCOS (*FRAMES*)

Un marco tampoco es un control propiamente dicho. Su función es la de englobar una serie de opciones (botones, cajas de texto, etc.…) con el fin de mantener una estructura ordenada de controles, separando unos de otros en función de las características del programa y del gusto del programador. En la Figura 32 pueden observarse dos marcos, en los que se sitúan dos botones.

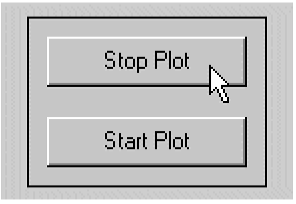


Figura 32. Marcos (***Frames***).

Ejemplo: El ejemplo dibuja un marco y dos botones, según se muestra en la Figura 32. Al pulsar el botón se imprime un mensaje en la ventana de MATLAB.

```
fig=gcf; clf;
ft_dir = uicontrol(gcf,...
   'Style','frame',...
   'Position',[30 30 120 85]);
pbstart = uicontrol(gcf,...
   'Style','push',...
   'Position',[40 40 100 25],...
   'String','Start Plot',...
   'CallBack',['disp(''Start Plot'')']);
```

```
pbstart = uicontrol(gcf,...
   'Style','push',...
   'Position',[40 80 100 25],...
   'String','Stop Plot',...
   'CallBack','disp(''Stop Plot'')');
```

8.5. Creación de menús

En MATLAB se pueden construir aplicaciones basadas en ventanas, con menús definidos por el usuario. En este apartado se describe cómo crear diferentes menús y submenús. Para crear los menús se utiliza la función ***uimenu***. El aspecto general del comando ***uimenu*** es el siguiente:

```
id_menu = uimenu(id_parent,...
                 'Propiedad1', 'valor1',...
                 'Propiedad2', 'valor2',...
                 'Propiedad3', 'valor3',...
                 'Otras Propiedades', 'Otros valores');
```

Normalmente una de las propiedades suele ser la propiedad ***CallBack***, aunque no siempre es necesario, ya que algunos menús no ejecutan ningún comando de MATLAB, sino que se encargan de desplegar nuevos submenús. Por lo tanto no es imprescindible definirla en todos los menús.

Para generar submenús basta con crear un menú nuevo, donde el ***id_parent*** sea el identificador del menú padre.

En la Figura 33 se muestra un detalle de los menús que componen una aplicación en particular (en este caso ***Microsoft Word 97***).

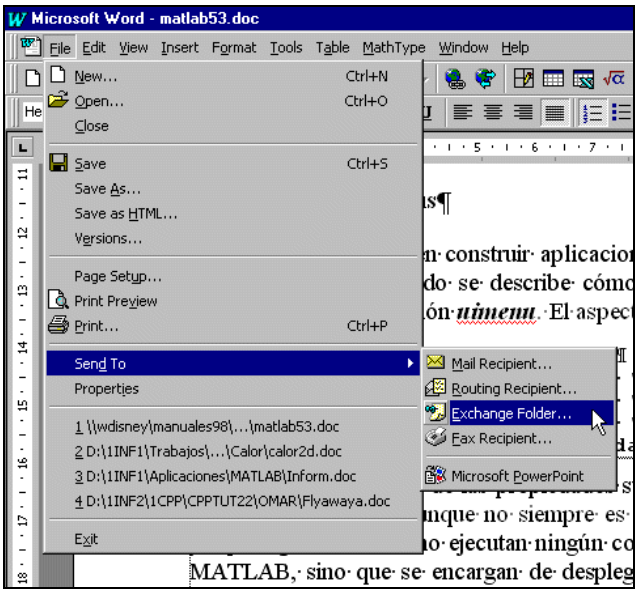


Figura 33. Detalle de los menús de ***Word 97***.

8.6. Descripción de las propiedades de los menús

A continuación se hace una descripción de las propiedades más importantes, con las que cuenta el comando ***uimenu***.

8.6.1. ACELERADOR (*ACCELERATOR*)

Esta propiedad es opcional y permite definir un carácter con el que activar el menú desde el teclado, sin necesidad de utilizar el ratón. Para activar el menú hay que teclear simultáneamente ALT + el carácter elegido.

```
uimenu(id_parent,...
           'Accelerator','carácter',...
      );
```

8.6.2. ACCIÓN A EFECTUAR POR EL MENÚ (*CALLBACK*)

Esta propiedad es muy importante, ya que determina la acción a realizar por MATLAB al actuar sobre el menú correspondiente.

```
uimenu(id_parent,...
           'CallBack','string',...
       );
```

8.6.3. CREACIÓN DE SUBMENUS (*CHILDREN*)

Permite crear submenús a partir de menús creados previamente.

```
uimenu(id_parent,...
           'Children',vector de handles,...
       );
```

8.6.4. MENÚ ACTIVADO/DESACTIVADO (*ENABLE*)

Esta propiedad permite modificar la posibilidad de acceso al menú correspondiente. A veces resulta interesante que un determinado menú o submenú esté inactivo, porque su acción no tenga sentido en algún momento de la ejecución del programa, (por ejemplo, ***Word*** no permite guardar un fichero, si no hay ninguno abierto). Los menús desactivados se muestran en color gris.

```
uimenu(id_parent,...
           'Enable','on'/'off',...
       );
```

8.6.5. NOMBRE DEL MENÚ (*LABEL*)

Esta propiedad establece el texto correspondiente del menú. Esta opción se puede combinar con la del acelerador, con el fin de señalar al usuario del programa, qué tecla debe pulsar para que se ejecute el comando. Así, anteponiendo el carácter (**&**) a la letra que se desee, ésta aparece subrayada. Esto no significa que este método sustituya al acelerador, solamente es una indicación para el usuario, que le indica la tecla con la que se activa el menú.

```
uimenu(id_parent,...
           'Label','string',...
       );
```

8.6.6. CONTROL DEL OBJETO PADRE (*PARENT*)

Esta opción especifica el ***id*** del objeto padre. Debe ir siempre en primer lugar. El padre de un ***uimenu*** es siempre una **figura** u otro ***uimenu***.

```
uimenu(id_parent,...
           'Parent',handle,...
       );
```

8.6.7. POSICIÓN DEL MENÚ (*POSITION*)

Esta propiedad se define mediante un escalar, que determina la posición relativa del menú. Un orden creciente en los menús, empezando a contar por uno, equivale a situarlos de izquierda a derecha en la barra de menús. En los submenús ese orden creciente significa que se colocan de arriba hacia abajo.

```
uimenu(id_parent,...
           'Position',scalar,...
       );
```

8.6.8. SEPARADOR (*SEPARATOR*)

Esta propiedad permite colocar una línea de separación encima del menú al que se aplica. Cuando se pone a ***on*** se traza la línea y cuando está en ***off*** no se coloca. Por defecto esta propiedad está en ***off***.

```
uimenu(id_parent,...
           'Separator', 'on'/'off',...
       );
```

8.6.9. VISIBLE (*VISIBLE*)

Puede tomar dos valores: ***on/off***. Indica si el menú es visible en la ventana o no.

```
uimenu(id_parent,...
           'Visible', 'on'/'off',...
       );
```

8.7. Ejemplo de utilización del comando *uimenu*

A continuación se muestra un ejemplo de utilización de los menús de usuario. El ejemplo de la Figura 34 consta de tres menús (***File***, ***Time*** y ***Curve***) en la barra de menús.

El menú ***File*** incluye tres submenús, ***Load***, ***Save*** y ***Exit***. El submenú ***Load*** permite cargar el fichero de variables ***data.mat***. El submenú ***Save*** salva las variables actuales en el fichero ***data.mat***, y el submenú ***Exit***, detiene la ejecución del programa. El menú ***Time*** permite seleccionar entre tres límites para la variable **t**, que es un vector. El menú ***Curve*** despliega a su vez dos submenús (***Sinus*** y ***Cosinus***), que trazan las gráficas de seno y coseno respectivamente, en función del tiempo seleccionado en el menú ***Time***. Después de elegir el tiempo hay que volver a elegir la curva que se desea visualizar. En la Figura 34 se muestra un aspecto general del programa.

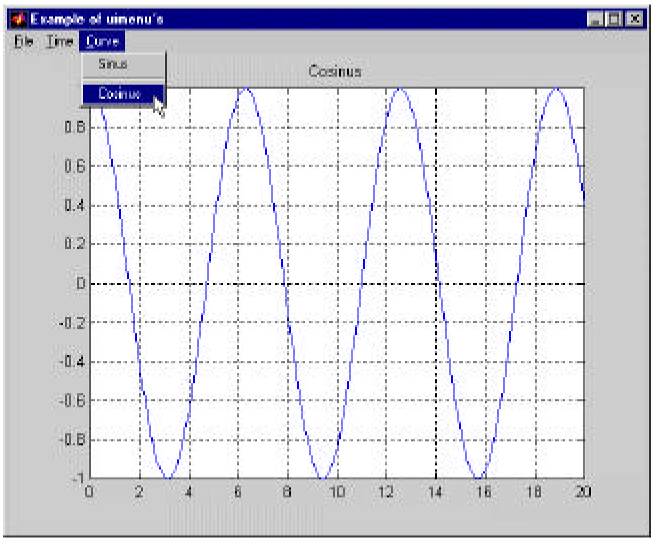


Figura 34. Ejemplo de menús programados por el usuario.

Para realizar esta aplicación, se crea en primer lugar la ventana del programa. A continuación se muestra cómo se debe crear una ventana sin ningún elemento estándar en la barra de menús. Los menús se irán incluyendo posteriormente.

```
% Crear una figura sin barra de menús
id_Fig = figure('Units','normalized',...
                'Numbertitle','off',...
                'Name','Ejemplo de uimenus',...
                'menubar','none');

% Creación de los diferentes menús
% Menú File
id_File = uimenu(id_Fig,'Label','&File',...
                  'Accelerator','f');
% Menú Time
id_Time = uimenu(id_Fig,'Label','&Time',...
                  'Accelerator','t');
```

```
% Menú Curve
id_Curve = uimenu(id_Fig,'Label','&Curve',...
                  'Accelerator','c');

% Creación de los diferentes submenús
% File
id_Load = uimenu(id_File,'Label','&Load',...
                  'Accelerator','L',...
                  'CallBack','load data.mat');
id_Save = uimenu(id_File,'Label','&Save',...
                  'Accelerator','s',...
                  'CallBack','save data.mat');
id_Exit = uimenu(id_File,'Label','E&xit',...
                  'Accelerator','x',...
                  'CallBack','close');
% Time
id_10 = uimenu(id_Time,'Label','10',...
                  'CallBack','t=0:.1:10;');
id_20 = uimenu(id_Time,'Label','20',...
                  'CallBack','t=0:.1:20;');
id_30 = uimenu(id_Time,'Label','30',...
                  'CallBack','t=0:.1:30;');
% Curve
id_Sinus = uimenu(id_Curve,'Label','Sinus',...
               'CallBack','plot(t,sin(t));grid;title(''Sinus'')');
id_Cosinus = uimenu(id_Curve,'Label','Cosinus',...
               'CallBack','plot(t,cos(t));grid;title(''Cosinus'')',...
               'Separator','on');
```

8.8. Menús contextuales (*uicontextmenu*)

Los ***menús contextuales*** son menús que se abren cuando el usuario realiza una determinada acción (normalmente clicar con el botón derecho) sobre un objeto de la figura.

Los menús contextuales se crean también con la función ***uimenu***, poniendo en ***on*** la propiedad ***UIContextMenu*** del objeto al que se quiere unir el menú contextual. para más información consultar el ***Help*** de MATLAB.

9. CONSTRUCCIÓN INTERACTIVA DE INTERFACES DE USUARIO (GUIDE)

MATLAB, a partir de la versión 5.0, ha incorporado un módulo llamado GUIDE (***Graphical User Interface Development Environment***) que permite crear de modo interactivo la interface de usuario, al modo de ***Visual Basic***, aunque todavía con unas posibilidades mucho más limitadas. En cualquier caso, si no es uno de los avances más importantes de MATLAB, si ha sido uno de los más agradecidos por los usuarios, que ya no tienen que escribir sin ayuda los ***callbacks*** del capítulo anterior.

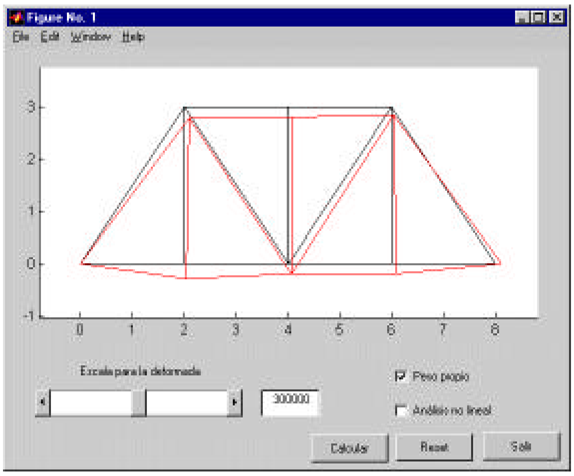


Figura 35. Interface de usuario creada con GUIDE.

En la Figura 35 se puede ver la interface de usuario de un programa de análisis de estructuras reticulares planas, desarrollada con GUIDE en unos pocos minutos (la interface de usuario, se entiende; el programa completo precisa lógicamente de más tiempo).

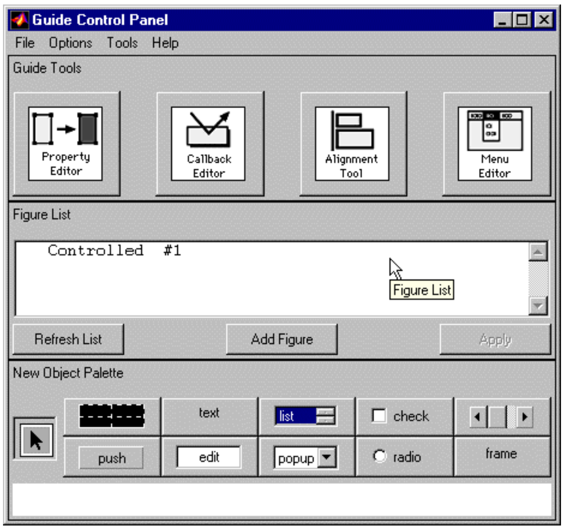


Figura 36. Ventana ***Guide Control Panel***.

9.1. Guide Control Panel

Como cualquier otro programa, GUIDE se ejecuta a partir de la línea de comandos de MATLAB, tecleando:

```
» guide
```

y pulsando ***Intro***. A continuación se abre la ventana ***Guide Control Panel*** (***GCP***), mostrada en la Figura 36. Se abre también una figura en blanco, sobre la que el diseñador deberá ir situando los distintos controles con el ratón, hasta terminar con el aspecto requerido, similar al de la aplicación mostrada en la Figura 35. La ventana ***GCP*** ocupa el papel central de la generación de los controles y menús de la interface de usuario. Dicha ventana contiene tres partes principales, dispuestas una debajo de otra.

La ***parte superior*** contiene cuatro grandes botones o iconos, correspondientes a los otros cuatro grandes módulos de GUIDE. De izquierda a derecha aparecen los iconos correspondientes a:

- el *Editor de Propiedades* (***Property Editor***),
- el *Editor de Llamadas* (***Callback Editor***),
- el *Editor de Alineamientos* (***Alignement Editor***) y
- el *Editor de Menús* (***Menu Editor***).

En lo sucesivo se hará referencia a estos módulos con la nomenclatura inglesa. Estos editores se pueden hacer visibles desde la línea de comandos de MATLAB (***propedit***, ***cbedit***, ***align*** y ***menuedit***), clicando en dichos iconos o seleccionándolos en el menú ***Tools*** de la ventana en la que se está haciendo el diseño (deberá estar en estado ***Controlled***).

La ***parte central*** de la ***GCP*** contiene la lista de ventanas o figuras de MATLAB controladas por GUIDE. En este caso sólo hay una, pero podría haber varias. Cada una de las figuras puede estar dos estados: *controlada* (***Controlled***) y *activa* (***Active***). Estos dos estados se corresponden con los modos "de diseño" y "de ejecución" de otras aplicaciones. Para que una figura pase de ***Controlled*** a ***Active*** hay que realizar dos acciones: 1.- Clicar sobre la línea correspondiente en la lista de figuras, con lo cual el mensaje en dicha lista pasa de un estado a otro, y, 2.-Clicar sobre el botón ***Apply***, con lo cual el estado seleccionado pasa a ser el estado real de la figura. Junto al botón ***Apply*** aparece otro botón llamado ***Add Figure*** que permite crear una nueva figura cuando se desee.

La ***parte inferior*** de la ***GCP*** contiene iconos correspondientes a bs elementos de interface de usuario (o controles) soportados por MATLAB 5.3, que son los siguientes (de izda a dcha y de arriba a abajo): ***axes***, ***text***, ***listbox***, ***checkbox***, ***slider***, ***pushbutton***, ***edit***, ***popupmenu***, ***radiobutton*** y ***frame***. Para crear uno de estos controles basta clicar sobre el icono correspondiente y luego ir a la figura en que se desee introducirlo (que deberá estar en estado ***Controlled***), clicar y arrastrar con el botón izquierdo del ratón pulsado para dar al control la posición y tamaño deseado. El nuevo control puede desplazarse y cambiarse de tamaño con ayuda del ratón, al igual que la propia ventana en la que ha sido situado Se puede observar que falta el icono correspondiente a ***togglebutton***. Para crear un botón de este tipo se crea un ***pushbutton*** y se cambia la propiedad ***Style*** a ***togglebutton***.

9.2. El Editor de Propiedades (*Property Editor*)

En un momento dado, sólo puede estar abierto un ***Property Editor*** (***PE***). Este editor tiene el aspecto mostrado en la Figura 37. Nótese que para que tenga este aspecto, los dos ***checkboxes*** (***Show Objecty Browser y Show Property List***) tienen que estar activados. En la parte superior del ***PE*** aparece una lista jerarquizada de los controles presentes en la figura (***Object Browser***). En este caso todos los controles son "hijos" de la figura nº 1, lo cual se muestra en el hecho de que todos aparezcan algo desplazados hacia la derecha. Para cada elemento se muestra el tipo (propiedad ***Style***) y un nombre (***EditText1*** para el elemento seleccionado). Este nombre se define por medio de la propiedad ***Tag***. Los nombres utilizados en este caso son los nombres por defecto propuestos por GUIDE.

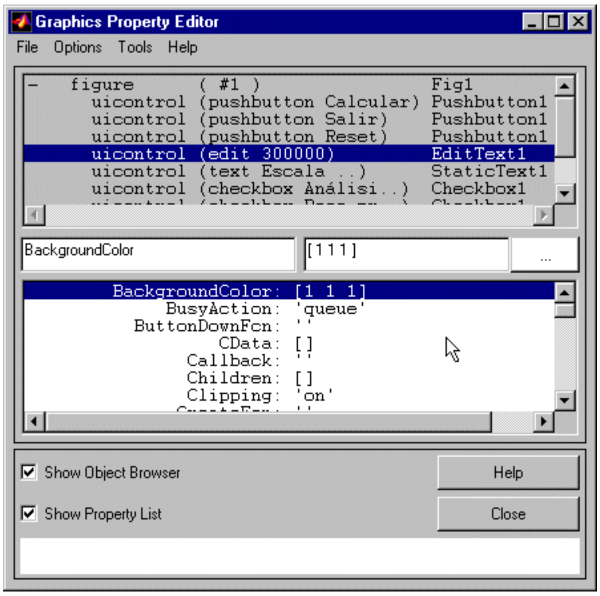


Figura 37. El Editor de Propiedades (***Property Editor***).

En la parte central del ***PE*** aparece una lista o relación de propiedades del objeto seleccionado en la parte superior. Las propiedades no se pueden editar directamente sobre esta lista: hay que seleccionar una propiedad y darle valor en la caja de texto que aparece inmediatamente encima de la lista, a la derecha (a la izquierda aparece el nombre de la propiedad). Para las propiedades que sólo pueden tomar ciertos valores, aparece una lista desplegable que los muestra y ayuda a elegirlos. En el caso de la Figura 37, como se trata de colores, aparece un botón con tres puntos, que da paso a un cuadro de diálogo en el que se puede elegir el color deseado en una paleta de colores. Para seleccionar una propiedad basta teclear sus primeras letras en la caja de texto que contiene el nombre y pulsar ***Intro***. El ***PE*** se encarga de buscar la propiedad deseada.

Es posible seleccionar varios objetos y establecer sus propiedades conjuntamente (las que tenga sentido hacerlo, como por ejemplo el color, el tamaño, etc.).

Es muy instructivo ver con calma las propiedades de un determinado objeto. Algunas pueden ser muy poco significativas en la mayor parte de los casos y otras se utilizan con mucha frecuencia: Entre las que casi siempre son las más importantes se pueden citar las siguientes:

<table>
<tr><td>String</td><td>El texto mostrado en casi todos los controles (botones, botones de opción o selección, cajas de texto, listas de selección, menús pop-up)</td></tr>
<tr><td>Label</td><td>Propiedad de uimenu que especifica el texto que aparece en el menú. Se puede utilizar el carácter & para especificar la tecla aceleradora, que aparecerá subrayada en el menú de la aplicación.</td></tr>
<tr><td>Tag</td><td>Un nombre interno para el objeto. No lo ve el usuario, pero se utiliza mucho en programación para localizar un determinado objeto. GUIDE asigna un Tag por defecto a cada objeto que se crea. Este nombre puede respetarse o ser sustituido por otro elegido por el usuario.</td></tr>
<tr><td>Style</td><td>La clase de objeto de que se trata (pushbutton | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu).</td></tr>
<tr><td>Position</td><td>Vector de cuatro elementos que indican la posición y tamaño del control [left, bottom, width, height]. Recuérdese que el origen está en la esquina inferior izquierda. Las unidades se expresan mediante la propiedad Units.</td></tr>
<tr><td>Extent</td><td>Vector de cuatro elementos que indica el tamaño del String de un objeto o elemento (etiqueta o texto mostrado).</td></tr>
<tr><td>Units</td><td>Unidades en que se miden las dimensiones: normalized miden desde (0,0) a (1.0,1.0). También pixels, inches, centimeters, y points (1/72 de una pulgada), que son unidades absolutas.</td></tr>
<tr><td>BackgroundColor</td><td>Vector de tres números entre 0 y 1.0 que indican las componentes RGB del color de fondo de un objeto.</td></tr>
<tr><td>ForegroundColor</td><td>Vector de tres números entre 0 y 1.0 que indican las componentes RGB del color del texto de un objeto.</td></tr>
<tr><td>Parent</td><td>El handle de la figura o control padre.</td></tr>
<tr><td>Children</td><td>Los handle de los controles hijos.</td></tr>
<tr><td>Enable</td><td>Si el control está activo o no, es decir si el usuario puede o no actuar sobre dicho control.</td></tr>
<tr><td>Visible</td><td>Si la figura o el control es visible o no.</td></tr>
<tr><td>FontName</td><td>El tipo de letra que se desea utilizar: 'Times New Roman', 'Arial', 'Courier New', etc.</td></tr>
<tr><td>FontSize</td><td>El tamaño de la letra en puntos (points).</td></tr>
<tr><td>FontWeight</td><td>Uno de estos valores: 'normal', 'ligth', 'demi' y bold'</td></tr>
<tr><td>UserData</td><td>Cualquier dato que el usuario quiera asociar con el control. No es utilizado por MATLAB. Los valores de UserData se pueden escribir con set() y leer con get().</td></tr>
<tr><td>Value</td><td>Valor asociado con algunos controles: posición del cursor en la barra de desplazamiento, valor de la propiedad max cuando están en on y min cuando están en off en los checkbuttons y radiobuttons; número ordinal (empezando por 1) del elemento seleccionado en las listbox y popupmenu. Las cajas de texto, los botones y los frames no tienen esta propiedad.</td></tr>
</table>

Para obtener ayuda sobre las propiedades de un objeto se puede proceder del siguiente modo. A partir del ***Help*** de MATLAB se abre el ***Matlab Help Desk*** en ***Netscape Navigator*** o ***Internet Explorer***; a continuación se clica en el enlace ***Handle Graphics Objects*** y se abre una página con la

jerarquía de objetos gráficos (ver Figura 25). Los elementos de esta jerarquía contienen ***enlaces*** a páginas de ayuda sobre cada uno de sus objetos (***figure***, ***axes***, ***uicontrol***, etc.). Cada una de esas páginas contiene información sobre las propiedades de cada elemento.

Algunos controles son un poco especiales. A continuación se hacen algunas consideraciones sobre ellos:

- Los controles ***listbox*** y ***porpupmenu*** contienen un conjunto de opciones. El texto de esas opciones se pueden almacenar en un ***cell array*** del espacio de trabajo base y luego la variable se introduce en la propiedad ***String*** del control (crea una copia llamada ***towork***).
- Para las barras de desplazamiento (***sliders***) los valores máximos y mínimos vienen dados por las propiedades ***max*** y ***min***. Los incrementos pequeño y grande vienen definidos por la propiedad ***SliderStep***, que es un vector de dos elementos.

9.3. El Editor de Llamadas (*Callback Editor*)

El ***Callback Editor (CE)*** es uno de los componentes más importantes de GUIDE, porque permite definir la forma en la que los distintos controles responden a las acciones del usuario (***eventos***). La ventana del ***CE*** se puede ver en la Figura 38.

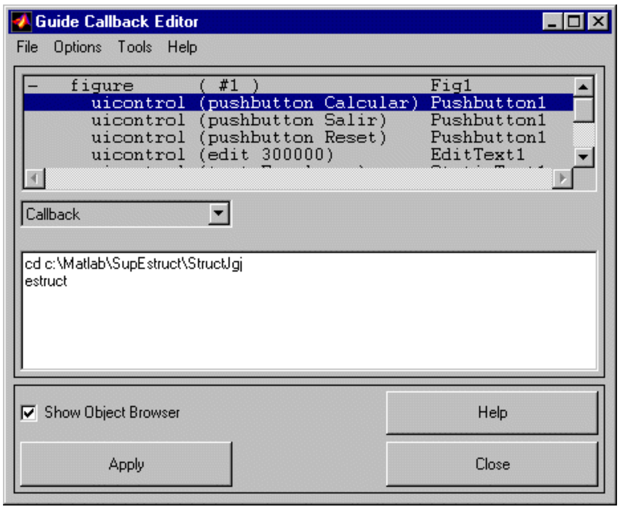


Figura 38. El ***Callback Editor***.

La respuesta que se desea obtener cuando el usuario realiza una determinada acción sobre un control se obtiene programando los ***callbacks***, que definen el código que se debe ejecutar en ese caso. Este código se asocia con el control por medio del ***CE***, en cuya parte superior pudde hacerse aparecer una lista de objetos (***Object Browser***). Seleccionando un objeto, en la caja de texto editable que aparece en el centro puede escribirse el código que se desea ejecutar al producirse el evento correspondiente. El evento se puede seleccionar en la lista desplegable que hay debajo del ***Object Browser***. Este código puede escribirse de dos maneras:

- Por medio de una cadena de caracteres que contiene los comandos de MATLAB (no es necesario poner las comillas simples externas porque el ***CE*** se ocupa de ello.
- Por medio de una llamada a una función que realice todas las operaciones necesarias. Este segundo método es claramente preferible al anterior, por dos motivos: 1) las cadenas de caracteres se interpretan cada vez que se ejecutan y esto es más lento que ejecutar una función, que se compila la primera vez que es llamada y se mantiene en memoria. 2) las funciones mantienen su propio espacio de trabajo, diferente del espacio de trabajo base y por tanto sin posibilidad de chocar o interferir con nombres de variables que estuvieran definidos en dicho espacio de trabajo base.

El ***CE*** permite definir ***callbacks*** correspondientes a eventos más especializados. Estos callbacks aparecen en el menú o lista desplegable que se ve en la Figura 38, encima de la caja de texto a la izquierda.

9.4. El Editor de Alineamientos (*Alignment Editor*)

El ***Alignment Editor (AE)*** es una herramienta auxiliar que permite que los controles situados en una ventana por medio de GUIDE, aparezcan uniformemente alineados o distribuidos. Su funcionamiento es bastante sencillo e intuitivo, por lo que no se darán muchos detalles aquí.

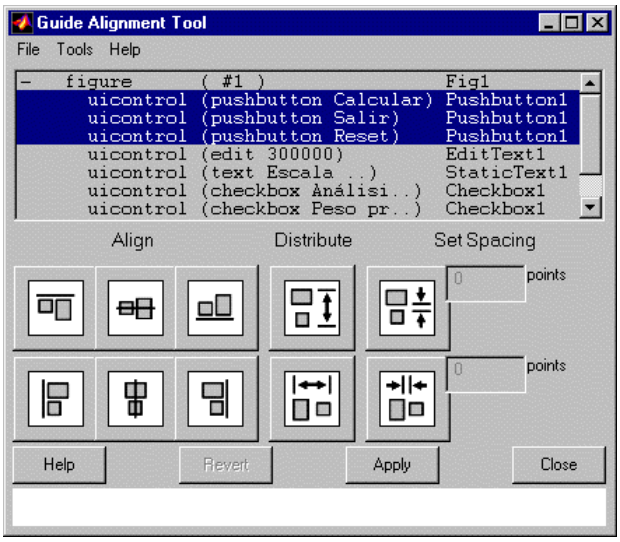


Figura 39. El ***Alignement Editor***.

La ventana del ***AE*** se muestra en la Figura 39. En la parte superior se muestra de nuevo la lista de objetos (***Object Browser***), en la que deberán estar seleccionados los objetos que se desea alinear o distribuir (para seleccionar varios objetos basta clicar sucesivamente sobre ellos manteniendo pulsada la tecla de ***Mayúsculas***).

Cuando se quiere alinear o distribuir varios objetos hay que tener en cuenta que el ***AE*** considera una caja "imaginaria" que comprende los objetos a ser alineados o distribuidos, y que realiza su operación de alineamiento o distribución dentro de esa caja.

9.5. El Editor de Menús (*Menu Editor*)

El ***Menu Editor (ME)*** es el último componente de GUIDE. Su ventana aparece en la Figura 40.

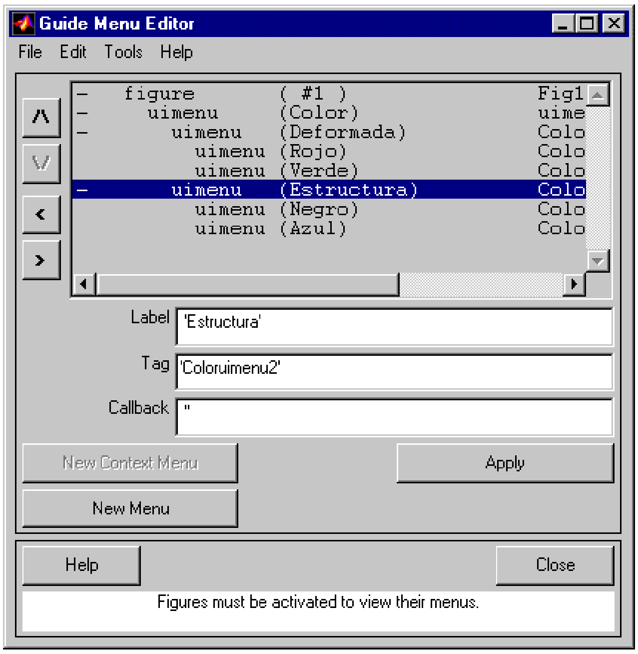


Figura 40. El ***Menu Editor***.

Hay que señalar que en toda ventana de MATLAB (***figure***) aparecen por defecto cuatro menús (***File***, ***Edit***, ***Windows*** y ***Help***). Los menús creados con el ***Menu Editor*** son menús adicionales. MATLAB permite hasta 4 niveles de menús.

En la parte superior del ***Menu Editor*** aparece una representación jerárquica de los menús. En este caso se ha introducido un menú ***Color***, con dos sub-menús, ***Deformada*** y ***Estructura***. Cada uno de ellos tiene dos colores opcionales.

A la izquierda de la lista de menús aparecen cuatro botones que permiten mover los menús por la jerarquía, aumentando o reduciendo su nivel en dicha jerarquía (*promote* y *demote*), cambiando su orden o pasándolos de un menú a otro.

Debajo de la jerarquía de menús del ***ME*** se muestran existen tres cajas de texto que permiten fijar el ***Label*** (lo que aparece como título del menú), el ***Tag*** (el nombre interno del menú) y el ***callback*** (la función o los comandos que se ejecutarán al elegir ese menú).

Para introducir un nuevo menú en la jerarquía basta pulsar el botón ***New Menu*** y establecer los datos que se deseen, situándolo con los botones en la posición deseada del menú deseado.

Pulsando ***Apply*** los datos de las cajas de texto se introducen en la jerarquía visible en la parte superior de la ventana.

9.6. Programación de *callbacks*

9.6.1. ALGUNAS FUNCIONES ÚTILES

Las siguientes funciones de MATLAB son muy útiles para la programación de interfaces de usuario gráficas interactivas:

gcf	(*get current figure*) devuelve el ***handle*** a la figura activa
gca	(*get current axes*) devuelve el ***handle*** de los ejes activos
gcbo	(*get callback object*) devuelve el ***handle*** del objeto sobre el que se ha producido el evento al que está tratando de responder el ***callback***
gcbf	(*get callback figure*) devuelve el ***handle*** de la figura sobre la que se ha producido el evento
findobj(gcbf, 'prop', 'propvalue')	devuelve el ***handle*** de un objeto de la figura activa cuya propiedad ***prop*** tiene el valor ***propvalue***. Los valores admitidos para la propiedad son los mismos que admite la función ***set***
get(handle, 'prop')	devuelve el valor de la propiedad ***prop*** en el objeto cuyo ***handle*** se pasa como primer argumento
set(handle, 'prop', 'propvalue')	establece el valor ***propvalue*** en la propiedad ***prop*** del objeto cuyo ***handle*** se pasa como primer argumento

Recuérdese que el ***handle*** es un número que identifica a cualquier objeto o elemento de la jerarquía gráfica de MATLAB. El ***handle*** se obtiene como valor de retorno en el momento de la creación del objeto y puede ser guardado en una variable para hacer referencia posteriormente a ese objeto. También puede ser hallado más tarde con alguna de las funciones vistas en este apartado, que devuelven el ***handle*** del objeto que cumple una determinada condición.

Hay que tener en cuenta que el ***handle*** no es un valor conocido de antemano (como puede ser por ejemplo la propiedad ***Tag*** de un objeto), sino que cambia cada vez que se ejecuta el programa. Por eso no se puede introducir en el código como constante numérica o alfanumérica, sino que hay que obtenerlo de algún modo en cada ejecución.

Los ***handles*** son el método utilizado por MATLAB para hacer referencia a los objetos de la interface gráfica de usuario y esto condiciona mucho el tipo de programación. En MATLAB no se puede utilizar la terminología típica ***objeto.propiedad*** de otros entornos de programación.

9.6.2. ALGUNAS TÉCNICAS DE PROGRAMACIÓN

Ya se ha comentado que es preferible gestionar los ***callbacks*** por medio de funciones que por medio de cadenas de caracteres que contengan comandos de MATLAB. Ésta es una de las primeras ideas que hay que tener en cuenta. Lo mismo se puede decir respecto a los ficheros de comandos ****.m*** que no son funciones: siempre será preferible utilizar funciones.

Recuérdese que las funciones de MATLAB tienen su propio espacio de trabajo y que la información necesaria para realizar su tarea les llega por uno de los siguientes caminos:

- por medio de los ***argumentos*** con los que se llama a la función,

- a través de ***variables globales*** (técnica que conviene evitar en la medida de lo posible),
- es también posible que la propia función se las arregle para encontrar de alguna manera la información que necesita, bien buscándola en algún recurso (un fichero, una variable accesible, ...) al que se tenga acceso, bien llamando a otras funciones que se la proporcionen.

En los ***callback*** de MATLAB hay que recurrir con mucha frecuencia a la tercera posibilidad: las funciones tienen que buscar la información que necesitan, de ordinario por medio de llamadas a las funciones vistas en el apartado anterior (***gcbo***, ***gcbf*** y ***findobj***).

Se hace referencia a las componentes u objetos de MATLAB por medio del ***handle***. Por ejemplo, para dibujar sobre unos ejes, para cambiar el texto que aparece sobre un botón, hace falta el ***handle*** del objeto sobre el que se quiere actuar. De forma análoga, para saber sobre qué objeto se ha producido el evento que da origen a un ***callback***, hay que localizar el ***handle*** de dicho objeto. Los ***handle*** se pueden determinar de las siguientes formas:

- Para obtener el ***handle*** de la figura en la que se ha producido un evento se utiliza la función ***gcbf***
- Para obtener el ***handle*** del objeto en el que se ha producido un evento se utiliza la función ***gcbo***
- Para obtener el ***handle*** de un objeto en el que se desea cambiar alguna propiedad se utiliza la función ***findobj(gcbf, 'prop', 'propvalue')***.
- MATLAB permite distintos tipos de evento sobre su jerarquía de objetos. La acción callback es la que está más directamente relacionada con la función del control (Clicar en un ***pushbutton***, cambiar el estado de un ***checkbutton*** o un ***radiobutton***, mover el cursor de una ***slider***, etc.). Hay otros eventos que disparan otras funciones ***callback***, tales como ***CreateFcn*** y ***DeleteFcn*** (que se ejecutan cuando se crea o destruye un objeto), ***WindowsButtonDownFcn***, ***WindowsButtonMotionFcn*** y ***WindowsButtonUpFcn*** (que se ejecutan cuando se pulsa, mueve o suelta un botón del ratón sobre una ventana) o ***ButtonDownFcn*** y ***KeyPressFcn*** (cuando se pulsa un botón del ratón o una tecla del teclado). Para más información mirar las propiedades de ***Figure***, ***Axes*** y ***Uicontrol*** con el ***Help Desk*** (HTML).

Es conveniente agrupar la gestión de los ***callbacks*** de varios componentes, objetos o controles en una misma función. Esta función puede calcular o recibir como argumento el objeto en el que se ha producido el evento, y luego con una sentencia ***switch*** (o con varios ***if... ifelse*** encadenados) ejecutar el código apropiado para el objeto sobre el que ha actuado el usuario y para la acción que éste ha realizado. Puede ser muy útil construir estas funciones utilizando ***sub-funciones***, es decir funciones definidas en el mismo fichero ****.m*** y que no pueden ser llamadas desde fuera de dicho fichero.

Para pasar valores de unas funciones a otras (o para compartir valores entre varias funciones de la interface de usuario) no es posible utilizar los argumentos. Se podría utilizar variables globales, pero ésta es una técnica considerada como peligrosa. Una solución es guardar dichos valores en la propiedad ***UserData*** que tienen las figuras y los controles de MATLAB. Se pueden almacenar valores con la función ***set()*** y recuperarlos con la función ***get()***.